SOMMERVILLE

ENGENHARIA DE SOFTWARE

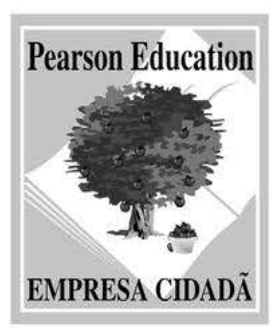
8ª EDIÇÃO

PEARSON
Addison Wesley

Site com material de apoio para professores e alunos

ENGENHARIA DE SOFTWARE

8ª EDIÇÃO



SOMMERVILLE

ENGENHARIA DE SOFTWARE

8ª EDIÇÃO

Tradução

Profa. dra. Selma Shin Shimizu Melnikoff
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais – PCS

Prof. dr. Reginaldo Arakaki
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais – PCS

Prof. dr. Edílson de Andrade Barbosa
Escola Politécnica da Universidade de São Paulo
Programa de Educação Continuada em Engenharia – PECE

Revisão técnica

Prof. dr. Kechi Hirama
Escola Politécnica da Universidade de São Paulo
Departamento de Engenharia de Computação e Sistemas Digitais – PCS

PEARSON Prentice Hall

São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha
Guatemala México Peru Porto Rico Venezuela

CÓPIA NÃO AUTORIZADA É CRIME
ABDR
ASSOCIAÇÃO BRASILEIRA DE DIREITOS REPROGRÁFICOS
RESPEITE O DIREITO AUTORAL
EDITORA AFILIADA

© 2007 by Pearson Education do Brasil
© Addison-Wesley Publishers Limited 1982, 1984
© 1989, 1995, 2001, 2004, 2007 by Pearson Education, Inc.
Tradução autorizada da edição original em inglês *Software Engineering*, 8ª ed., de SOMMERVILLE, Ian publicada pela Pearson Education Inc, sob o selo Addison-Wesley.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Gerente editorial: Roger Trimer
Editora sênior: Sabrina Cairo
Editor de desenvolvimento: Marco Pace
Editora de texto: Sheila Fabre
Preparação: Alessandra Miranda de Sá
Revisão: Aiko Nishijima
Capa: Celso Blanes
Editoração Eletrônica: Figurativa Editorial MM Ltda

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Sommerville, Ian
Engenharia de software, 8ª edição / Ian Sommerville; tradução: Selma Shin Shimizu Melnikoff, Reginaldo Arakaki, Edílson de Andrade Barbosa; revisão técnica: Kechi Kirama. -- 8 ª ed. -- São Paulo : Pearson Addison-Wesley, 2007.

Título original: Software Engineering, eighth edition.

ISBN 978-85-88639-28-7

1. Engenharia de software I. Título

07-1842 CDD-005.1

Índices para catálogo sistemático:

1. Engenharia de software 005.1

2007
Direitos exclusivos para a língua portuguesa cedidos à
Pearson Education do Brasil, uma empresa do grupo Pearson Education.
Av. Ermano Marchetti, 1435
CEP: 05038-001 – Lapa – São Paulo – SP
Fone (11) 2178-8686 – Fax (11) 3611-0444
e-mail: vendas@pearsoned.com

SUMÁRIO

Prefácio xi

PARTE 1 VISÃO GERAL **1**

CAPÍTULO 1 Introdução 3
1.1 Perguntas mais freqüentes sobre engenharia de software 4
1.2 Responsabilidade profissional e ética 10
Pontos-chave 12
Leituras sugeridas 12
Exercícios 13

CAPÍTULO 2 Sistemas sociotécnicos 14
2.1 Propriedades emergentes de sistemas 16
2.2 Engenharia de sistemas 17
2.3 Organizações, pessoas e sistemas de computadores 23
2.4 Sistemas legados 25
Pontos-chave 27
Leituras sugeridas 27
Exercícios 27

CAPÍTULO 3 Sistemas críticos 29
3.1 Sistema crítico de segurança simples 30
3.2 Confiança no sistema 32
3.3 Disponibilidade e confiabilidade 34
3.4 Segurança 37
3.5 Proteção 38
Pontos-chave 40
Leituras sugeridas 40
Exercícios 40

CAPÍTULO 4 Processos de software 42
4.1 Modelos de processo de software 43
4.2 Iteração de processo 47
4.3 Atividades de processo 49
4.4 O Rational Unified Process 54
4.5 Engenharia de Software Auxiliada por Computador 56
Pontos-chave 59
Leituras sugeridas 59
Exercícios 59

CAPÍTULO 5 Gerenciamento de projetos 61
5.1 Atividades de gerenciamento 62
5.2 Planejamento de projeto 63
5.3 Cronograma do projeto 65
5.4 Gerenciamento de riscos 69
Pontos-chave 74
Leituras sugeridas 74
Exercícios 74

PARTE 2 REQUISITOS 77

CAPÍTULO 6 Requisitos de software 79
6.1 Requisitos funcionais e não funcionais 80
6.2 Requisitos de usuário 85
6.3 Requisitos de sistema 87
6.4 Especificação de interface 89
6.5 Documento de requisitos de software 91
Pontos-chave 93
Leituras sugeridas 94
Exercícios 94

CAPÍTULO 7 Processos de engenharia de requisitos 95
7.1 Estudos de viabilidade 97
7.2 Elicitação e análise de requisitos 97
7.3 Validação de requisitos 105
7.4 Gerenciamento de requisitos 107
Pontos-chave 110
Leituras sugeridas 111
Exercícios 111

CAPÍTULO 8 Modelos de sistema 112
8.1 Modelos de contexto 113
8.2 Modelos de comportamento 114
8.3 Modelos de dados 117
8.4 Modelos de objetos 120
8.5 Métodos estruturados 124
Pontos-chave 126
Leituras sugeridas 126
Exercícios 126

CAPÍTULO 9 Especificação de sistemas críticos 128
9.1 Especificação dirigida a riscos 129
9.2 Especificação de segurança 134
9.3 Especificação de proteção 136
9.4 Especificação de confiabilidade de software 137
Pontos-chave 141
Leituras sugeridas 142
Exercícios 142

CAPÍTULO 10 Especificação formal 144
10.1 Especificação formal no processo de software 145
10.2 Especificação de interface de subsistema 147
10.3 Especificação de comportamento 152
Pontos-chave 156
Leituras sugeridas 156
Exercícios 156

PARTE 3 PROJETO 159

CAPÍTULO 11 Projeto de arquitetura ... 161
11.1 Decisões de projeto de arquitetura ... 163
11.2 Organização de sistema ... 164
11.3 Estilos de decomposição modular ... 168
11.4 Modelos de controle ... 170
11.5 Arquiteturas de referência ... 173
Pontos-chave ... 175
Leituras sugeridas ... 175
Exercícios ... 175

CAPÍTULO 12 Arquiteturas de sistemas distribuídos ... 177
12.1 Arquiteturas de multiprocessadores ... 179
12.2 Arquiteturas cliente–servidor ... 179
12.3 Arquiteturas de objetos distribuídos ... 183
12.4 Computação interorganizacional distribuída ... 188
Pontos-chave ... 192
Leituras sugeridas ... 192
Exercícios ... 192

CAPÍTULO 13 Arquiteturas de aplicações ... 194
13.1 Sistemas de processamento de dados ... 196
13.2 Sistemas de processamento de transações ... 197
13.3 Sistemas de processamento de eventos ... 202
13.4 Sistemas de processamento de linguagens ... 204
Pontos-chave ... 206
Leituras sugeridas ... 206
Exercícios ... 206

CAPÍTULO 14 Projeto orientado a objetos ... 208
14.1 Objetos e classes de objetos ... 210
14.2 Processo de projeto orientado a objetos ... 213
14.3 Evolução de projeto ... 222
Pontos-chave ... 223
Leituras sugeridas ... 223
Exercícios ... 224

CAPÍTULO 15 Projeto de software de tempo real ... 225
15.1 Projeto de sistemas ... 227
15.2 Sistemas operacionais de tempo real ... 229
15.3 Sistemas de monitoração e controle ... 231
15.4 Sistemas de aquisição de dados ... 235
Pontos-chave ... 238
Leituras sugeridas ... 238
Exercícios ... 238

CAPÍTULO 16 Projeto de interface com o usuário ... 240
16.1 Tópicos de projeto ... 243
16.2 Processo de projeto de UI ... 249
16.3 Análise de usuário ... 250
16.4 Prototipação de interface com o usuário ... 253
16.5 Avaliação de interface ... 254
Pontos-chave ... 255
Leituras sugeridas ... 256
Exercícios ... 256

PARTE 4 DESENVOLVIMENTO 257

CAPÍTULO 17 Desenvolvimento rápido de software ... 259
17.1 Métodos ágeis ... 262
17.2 *Extreme programming* ... 263
17.3 Desenvolvimento rápido de aplicações ... 268
17.4 Prototipação de software ... 270
Pontos-chave ... 273
Leituras sugeridas ... 273
Exercícios ... 273

CAPÍTULO 18 Reuso de software ... 275
18.1 O panorama do reuso ... 277
18.2 Design patterns ... 279
18.3 Reuso baseado em geradores ... 281
18.4 Frameworks de aplicações ... 282
18.5 Reuso do sistema de aplicações ... 283
Pontos-chave ... 289
Leituras sugeridas ... 289
Exercícios ... 289

CAPÍTULO 19 Engenharia de software baseada em componentes ... 291
19.1 Componentes e modelos de componentes ... 293
19.2 Processo CBSE ... 298
19.3 Composição de componentes ... 299
Pontos-chave ... 304
Leituras sugeridas ... 304
Exercícios ... 305

CAPÍTULO 20 Desenvolvimento de sistemas críticos ... 306
20.1 Processos confiáveis ... 308
20.2 Programação confiável ... 309
20.3 Tolerância a defeitos ... 314
20.4 Arquiteturas tolerantes a defeitos ... 319
Pontos-chave ... 321
Leituras sugeridas ... 321
Exercícios ... 321

CAPÍTULO 21 Evolução de software ... 323
21.1 Dinâmica da evolução de programas ... 324
21.2 Manutenção de software ... 326
21.3 Processo de evolução ... 329
21.4 Evolução de sistemas legados ... 333
Pontos-chave ... 337
Leituras sugeridas ... 337
Exercícios ... 337

PARTE 5 VERIFICAÇÃO E VALIDAÇÃO 339

CAPÍTULO 22 Verificação e validação ... 341
22.1 Planejamento de verificação e validação ... 344
22.2 Inspeções de software ... 345
22.3 Análise estática automatizada ... 348
22.4 Verificação e métodos formais ... 350
Pontos-chave ... 353

Leituras sugeridas 354
Exercícios 354

CAPÍTULO 23 Teste de software 355
23.1 Teste de sistemas 357
23.2 Teste de componentes 361
23.3 Projeto de casos de teste 364
23.4 Automação de testes 371
Pontos-chave 372
Leituras sugeridas 372
Exercícios 373

CAPÍTULO 24 Validação de sistemas críticos 374
24.1 Validação de confiabilidade 375
24.2 Garantia de segurança 379
24.3 Avaliação de proteção 384
24.4 Casos de segurança e confiabilidade 385
Pontos-chave 387
Leituras sugeridas 387
Exercícios 388

PARTE 6 GERENCIAMENTO 389

CAPÍTULO 25 Gerenciamento de pessoal 391
25.1 Seleção de pessoal 392
25.2 Motivação de pessoas 394
25.3 Gerenciamento de grupos 396
25.4 O Modelo de Maturidade de Capacitação de Pessoal 402
Pontos-chave 403
Leituras sugeridas 403
Exercícios 403

CAPÍTULO 26 Estimativa de custo de software 405
26.1 Produtividade de software 407
26.2 Técnicas de estimativa 410
26.3 Modelagem algorítmica de custos 411
26.4 Duração de projeto e seleção de pessoal 420
Pontos-chave 421
Leituras sugeridas 422
Exercícios 422

CAPÍTULO 27 Gerenciamento de qualidade 423
27.1 Qualidade de processo e de produto 425
27.2 Garantia de qualidade e padrões 426
27.3 Planejamento de qualidade 430
27.4 Controle de qualidade 431
27.5 Medições e métricas de software 432
Pontos-chave 437
Leituras sugeridas 437
Exercícios 438

CAPÍTULO 28 Aprimoramento de processo 439
28.1 Qualidade de processo e de produto 441
28.2 Classificação de processos 442
28.3 Medição de processos 443
28.4 Análise e modelagem de processos 444

28.5 Mudança de processos ... 447
28.6 Framework CMMI de aprimoramento de processos ... 448
Pontos-chave ... 453
Leituras sugeridas ... 453
Exercícios ... 453

CAPÍTULO 29 Gerenciamento de configurações ... 455
29.1 Planejamento de gerenciamento de configurações ... 457
29.2 Gerenciamento de mudanças ... 459
29.3 Gerenciamento de versões e releases ... 461
29.4 Construção de sistemas ... 465
29.5 Ferramentas CASE para gerenciamento de configurações ... 466
Pontos-chave ... 469
Leituras sugeridas ... 470
Exercícios ... 470

PARTE 7 TECNOLOGIAS EMERGENTES 471

CAPÍTULO 30 Engenharia de proteção ... 473
30.1 Conceitos de proteção ... 475
30.2 Gerenciamento de riscos de proteção ... 476
30.3 Projeto para proteção ... 480
30.4 Capacidade de sobrevivência de sistemas ... 486
Pontos-chave ... 489
Leituras sugeridas ... 489
Exercícios ... 489

CAPÍTULO 31 Engenharia de software orientada a serviços ... 491
31.1 Serviços como componentes reusáveis ... 493
31.2 Engenharia de serviços ... 496
31.3 Desenvolvimento de software como serviços ... 502
Pontos-chave ... 507
Leituras sugeridas ... 507
Exercícios ... 508

CAPÍTULO 32 Desenvolvimento de software orientado a aspectos ... 509
32.1 Separação de assuntos ... 510
32.2 Aspectos, pontos de junção e de corte ... 513
32.3 Engenharia de software com aspectos ... 516
Pontos-chave ... 523
Leituras sugeridas ... 524
Exercícios ... 524

Glossário ... 525
Referências ... 531
Índice remissivo ... 543
Índice onomástico ... 551

PREFÁCIO

A primeira edição deste livro sobre engenharia de software foi publicada há mais de vinte anos. Ela foi escrita em um terminal 'burro' ligado a um antigo minicomputador (um PDP-11) que, provavelmente, custou cerca de 50 mil dólares. Escrevi esta edição usando um laptop com tecnologia wireless que custa menos de dois mil dólares e é muito mais potente do que aquele PDP-11. O software então era, principalmente, para mainframes, mas os computadores pessoais estavam começando a se tornar disponíveis. Nenhum de nós imaginou o quanto eles invadiriam nossa vida nem quanto eles mudariam o mundo.

As mudanças no hardware ao longo dos últimos vinte e poucos anos foram absolutamente extraordinárias e pode-se notar que as mudanças nos softwares foram igualmente significativas. Certamente, a capacidade de criar grandes e complexos sistemas aumentou. Os serviços e a infra-estrutura nacionais — energia, comunicações e transporte — contam amplamente com sistemas de computadores muito complexos e confiáveis. Para criar sistemas de negócios, existe uma sopa de letrinhas tecnológica — J2EE, .NET, EJB, SAP, BPEL4WS, SOAP, CBSE — que permite a grandes sistemas baseados na Web serem implantados muito mais rapidamente do que era possível no passado.

No entanto, embora pareça que tenha havido muita mudança nos últimos vinte anos, quando olhamos além das tecnologias específicas para processos fundamentais de engenharia de software, muitas coisas permaneceram inalteradas. Reconhecemos há vinte anos que o modelo cascata do processo de software tinha sérios problemas, mas um estudo publicado em 2003 em *IEEE software* mostrou que mais de 40 por cento das empresas usam essa abordagem. O teste é ainda uma técnica dominante de verificação de programas, embora outras técnicas, como inspeções, tenham sido usadas mais eficientemente a partir da metade da década de 1970. As ferramentas CASE, apesar de serem utilizadas atualmente com UML, são essencialmente editores de diagramas com alguma funcionalidade de verificação e geração de código.

Os métodos e técnicas atuais de engenharia de software tornaram-nos muito mais capazes de criar sistemas grandes e complexos. No entanto, ainda existem muitos projetos que atrasam, ultrapassam o orçamento e não produzem software que atenda às necessidades do cliente. Enquanto estava escrevendo a 7ª edição, uma pesquisa governamental no Reino Unido informou sobre o projeto de um sistema nacional a ser utilizado em tribunais para julgar transgressores relativamente menores. O custo desse sistema foi estimado em £156 milhões e foi programado para ser entregue em 2001. Em 2004, os custos tinham subido para £390 milhões e o sistema ainda não era de todo operacional. Portanto, existe uma necessidade urgente do ensino da engenharia de software.

Nos últimos anos, os avanços mais importantes na engenharia de software foram o aparecimento da UML como padrão para a descrição de sistemas orientados a objetos e o desenvolvimento de métodos ágeis como o *extreme programming*. Os métodos ágeis estão voltados ao desenvolvimento rápido de sistemas, envolvem explicitamente os usuários na equipe de desenvolvimento e reduzem documentos e burocracia no processo de software. Apesar da reclamação de alguns críticos, penso que essas abordagens incorporam boas práticas de engenharia de software. Esses métodos possuem um processo bem definido, dão atenção à especificação do sistema e aos requisitos de usuário e possuem altos padrões de qualidade.

Contudo, esta revisão não se tornou um texto sobre métodos ágeis. Em vez disso, focalizo os processos básicos de engenharia de software — especificação, projeto, desenvolvimento, verificação, validação e gerenciamento. Você deve compreender esses processos e as técnicas associadas para decidir se os métodos ágeis constituem uma estratégia de desenvolvimento mais apropriada para seu caso e como adaptar e alterar os métodos para atender às suas necessidades específicas. Um tema que permeia este livro são os sistemas críticos — cuja falha tem conseqüências graves e nos quais a confiança é grande. Em cada parte do livro, explico técnicas específicas de engenharia de software relevantes para a engenharia de sistemas críticos.

Os livros refletem, inevitavelmente, as opiniões e os preconceitos dos autores. Alguns leitores discordarão das minhas opiniões e de minha seleção de material. Tal discordância é um reflexo saudável da diversidade da disciplina e é essencial para sua evolução. Todavia, espero que todos os engenheiros de software e estudantes de engenharia de software possam encontrar algo de interessante nesta obra.

Estrutura do livro

O conteúdo deste livro está baseado nos processos fundamentais de engenharia de software e é constituído de sete partes. As primeiras seis focalizam os processos de software e a parte final explica algumas novas e importantes tecnologias de engenharia de software.

Parte 1: apresenta a engenharia de software, posicionando-a em um contexto mais amplo de sistema, e apresenta as noções sobre processos e gerenciamento de engenharia de software.

Parte 2: aborda os processos, as técnicas e os produtos associados à engenharia de requisitos. Inclui uma discussão sobre requisitos de software, modelagem de sistema, especificação formal e técnicas de especificação de confiança.

Parte 3: esta parte está dedicada ao projeto de software e aos processos de projeto. Três dos seis capítulos abordam o importante tópico de arquiteturas de software. Outros tópicos incluem o projeto orientado a objetos, o projeto de sistemas de tempo real e o projeto de interface com o usuário.

Parte 4: descreve uma série de abordagens para desenvolvimento, entre elas métodos ágeis, reuso de software, CBSE e desenvolvimento de sistemas críticos. Como a mudança no sistema constitui, atualmente, grande parte do desenvolvimento, incluí um material sobre evolução e manutenção de software nesta parte.

Parte 5: apresenta as técnicas para verificação e validação de software. Inclui capítulos sobre V & V estáticas, teste e validação de sistemas críticos.

Parte 6: esta parte abrange tópicos de gerenciamento: gerenciamento de pessoas, estimativa de custos, gerenciamento de qualidade, aprimoramento de processos e gerenciamento de configuração.

Parte 7: a parte final inclui três capítulos dedicados às novas e importantes tecnologias que já começaram a ser usadas — os capítulos sobre engenharia de segurança, engenharia de software orientada a serviços e desenvolvimento de software orientado a aspectos.

Na introdução de cada parte, explico a estrutura e a organização com mais detalhes.

Mudanças em relação à 7ª edição

Esta nova edição deve ser considerada mais como uma atualização do que como uma revisão completa do livro. Elaborei-a para ser *completamente* compatível com a 7ª edição, mas incluí uma nova seção sobre tecnologias emergentes. São explicados os desenvolvimentos recentes que considero significativos para o futuro da engenharia de software. Essa seção inclui três capítulos adicionais:

30. "Engenharia de proteção", no qual explico questões sobre como garantir que seu software seja seguro e resista a ataques externos.
31. "Engenharia de software orientada a serviços", no qual descrevo novas abordagens para desenvolvimento de aplicações usando serviços Web reusáveis.
32. "Desenvolvimento de software orientado a aspectos", no qual apresento uma nova técnica de desenvolvimento de software baseada na separação de assuntos.

Como os outros capítulos deste livro ainda são atuais e relevantes, não os modifiquei, com exceção de algumas pequenas alterações com referências ao novo material dos capítulos 30 a 32.

Público-alvo

Este livro destina-se aos estudantes dos cursos de graduação e pós-graduação e a engenheiros de software que atuam no comércio e na indústria. Ele pode ser usado em cursos de engenharia de software em geral ou em outros cursos, como os de programação avançada, especificação de software, projeto ou gerenciamento de software. Os engenheiros de software da área industrial podem achar o livro útil para leitura geral ou como meio de atualizar seus conhecimentos sobre tópicos específicos, como engenharia de requisitos, projeto de arquitetura, desenvolvimento de sistemas confiáveis e aprimoramento de processo. Sempre que possível, apresentei uma visão prática dos exemplos no texto, a fim de refletir os tipos de aplicações que os engenheiros de software devem desenvolver.

Utilização do livro como livro-texto

Este livro é amplamente utilizado em uma série de cursos de engenharia de software e, se você já usa a edição anterior, achará esta nova completamente compatível com ela. Deixei os capítulos 1 a 29 inalterados propositadamente. Se você os usa para ministrar aulas, não há necessidade de alterar nenhum material complementar ou das atividades do curso. Os novos capítulos são independentes e você pode utilizar um ou mais para dar aos estudantes um entendimento dos novos avanços no assunto.

Elaborei o livro de maneira que possa ser utilizado em três tipos de cursos de engenharia de software:

1. *Cursos introdutórios gerais de engenharia de software*: para estudantes que não possuem experiência anterior em engenharia de software, é possível começar com a seção introdutória e selecionar capítulos de outras seções do livro. Isso fornecerá aos estudantes uma visão geral do assunto e possibilitará um estudo mais detalhado para os alunos mais interessados. Se a abordagem do curso for baseada em projetos, os capítulos iniciais fornecem material suficiente para permitir que os alunos sejam iniciados nos projetos, consultando os capítulos posteriores e informações adicionais à medida que o trabalho avança.
2. *Cursos introdutórios ou intermediários sobre tópicos específicos de engenharia de software*: o livro oferece material de apoio a cursos sobre especificação de requisitos de software, projeto de software, gerenciamento de engenharia de software, desenvolvimento de sistemas confiáveis e evolução de software. Cada parte do livro pode funcionar de modo independente para cursos introdutório ou intermediário sobre esses assuntos.
3. *Cursos mais avançados sobre tópicos específicos de engenharia de software*: os capítulos podem formar a base de um curso específico de software, mas devem ser complementados com leitura adicional que explore o tópico mais detalhadamente.

As vantagens de um texto genérico como este é que ele pode ser utilizado em vários cursos relacionados. O texto pode ser utilizado em um curso introdutório de engenharia de software e em cursos de especificação, projeto e sistemas críticos. Em cursos sobre engenharia de software baseada em componentes e engenharia de sistemas o livro é usado junto com artigos adicionais distribuídos aos alunos. Com um único texto que apresente uma visão consistente do assunto, os alunos não precisam comprar vários livros. Além disso, cada capítulo possui:

- um conjunto de objetivos claramente definidos na primeira página;
- uma lista dos pontos-chave abordados no capítulo;
- sugestões de leitura adicional — livros atualmente disponíveis ou artigos facilmente encontráveis;
- exercícios, que incluem exercícios de projeto.

O projeto Software Engineering Body of Knowledge (http://www.swebok.org) foi estabelecido para definir as principais áreas de conhecimento técnico relevantes para os engenheiros de software profissionais. Elas são organizadas em dez categorias: requisitos, projeto, criação, teste, manutenção, gerenciamento de configuração, gerenciamento, processo, ferramentas e métodos e qualidade. Embora fosse impossível abranger todas as áreas de conhecimento propostas pelo projeto SWEBOK em um único livro, todas as principais áreas são explicadas aqui.

Companion Website

No Companion Website deste livro (http://www.aw.com/sommerville_br), professores e alunos obtêm materiais de apoio para o aprendizado da disciplina.

Para os professores estão disponíveis:

- apresentações em PowerPoint com o resumo de todos os capítulos do livro;
- sugestões de projetos.

Para os alunos:

- sugestões de leitura adicional e links para recursos Web relacionados a cada capítulo do livro;
- soluções dos exercícios escolhidos de cada capítulo.

No site Web do autor, o leitor também encontra, em inglês, sugestões de leitura adicional e estudos de caso. O endereço desse site é: http://www.software-engin.com.

Tem sido minha política, tanto na edição anterior quanto nesta, mencionar a menor quantidade possível de links no livro. A razão para isso é que esses links estão constantemente sujeitos a mudanças e, uma vez impressos, torna-se impossível atualizá-los. Assim, o site Web deste livro inclui um grande número de links para recursos e materiais relacionados a engenharia de software. Se você tentar acessá-los e tiver problemas, por favor, me informe e eu os atualizarei.

Agradeço seus comentários e sugestões construtivas sobre o livro e o site Web. Você pode me contatar pelo e-mail ian@software-engin.com. Recomendo que inclua o código [SE8] na linha de assunto do e-mail para garantir que meu filtro de spam não rejeite acidentalmente sua mensagem. Lamento, mas infelizmente não tenho tempo para ajudar todos os alunos com suas tarefas de casa; portanto, por favor, não me perguntem como resolver qualquer um dos problemas apresentados neste livro.

Agradecimentos

Várias pessoas contribuíram ao longo dos anos para o aprimoramento deste livro, e eu gostaria de agradecer a todos (revisores, alunos e usuários que me enviaram mensagens) que comentaram as edições anteriores e deram sugestões construtivas para mudanças. A equipe editorial e de produção da Pearson Education na Inglaterra e Estados Unidos forneceu apoio e ajuda, e produziu o livro em tempo recorde. Agradeço a Simon Plumtree, Mary Lince, Ros Woodward, Keith Mansfield, Patty Mahtani, Daniel Rausch, Carol Noble e Sharon Burkhardt pela ajuda e apoio.

Enquanto finalizava este livro, estava prestes a deixar a Lancaster University para enfrentar novos desafios na St. Andrews University, na Escócia. Gostaria de agradecer a todos os meus colegas atuais e antigos na Lancaster pelo apoio e estímulo durante os anos em que a engenharia de software evoluiu.

Finalmente, gostaria de agradecer à minha família, que tolerou não apenas a minha ausência enquanto me dedicava ao livro mas também a minha frustração, quando as palavras não surgiam. Um enorme 'obrigado' à minha esposa Anne e às minhas filhas, Ali e Jane, por toda a ajuda e apoio.

IAN SOMMERVILLE
FEVEREIRO DE 2006

PARTE 1 VISÃO GERAL

A estrutura básica deste livro segue os processos de software essenciais de especificação, projeto, desenvolvimento, verificação e validação e gerenciamento. No entanto, antes de mergulhar imediatamente nos tópicos, incluí esta seção para que você possa ter um panorama amplo da disciplina. Os capítulos que constituem esta parte são os seguintes:

- O Capítulo 1 é uma introdução geral à engenharia de software. Para torná-lo acessível e fácil de entender, eu o organizei usando a estrutura de perguntas e respostas, tal como 'o que é engenharia de software'. Neste capítulo também discuto os temas *profissionalismo* e *ética*.
- O Capítulo 2 apresenta os sistemas sociotécnicos, um tópico que acredito ser absolutamente essencial para engenheiros de software. O software nunca é usado isoladamente, mas sempre como parte de um sistema mais amplo, incluindo hardware, pessoas e, freqüentemente, organizações. Eles afetam profundamente os requisitos e a operação do software. Neste capítulo, são abordadas as propriedades essenciais de sistema, os processos de engenharia de sistema e algumas formas de como os interesses organizacionais e humanos afetam os sistemas de software.
- O Capítulo 3 explica os 'sistemas críticos'. Esses sistemas são aqueles cuja falha causa graves conseqüências técnicas, econômicas e humanas, e nos quais a segurança, a proteção e a disponibilidade são requisitos-chave. Capítulos sobre aspectos de sistemas críticos estão incluídos em cada parte do livro. Neste capítulo, apresento também o primeiro dos estudos de caso do livro — o software para uma bomba de insulina, usada no tratamento de pacientes diabéticos.
- Os três primeiros capítulos definem o cenário da engenharia de software, e o Capítulo 4 estende esse assunto apresentando o processo de software e seus modelos. Apresentei processos básicos de engenharia de software, o assunto do livro, nesse capítulo. Também explico brevemente o Processo Unificado Racional (Rational Unified Process), que é dirigido ao desenvolvimento de sistemas orientados a objetos. A seção final do capítulo explica como os processos de software podem ser apoiados por ferramentas automatizadas de software.
- O Capítulo 5 aborda o gerenciamento de projetos. Este tópico faz parte de todos os projetos profissionais de desenvolvimento e eu descrevo o planejamento básico de projetos, programação e estimativa de riscos aqui. Os estudantes de um curso de engenharia de software envolvidos em um projeto devem encontrar as informações necessárias para elaborar gráficos de barras para programação de projeto e alocação de recursos.

1

Introdução

Objetivos

Os objetivos deste capítulo são apresentar a engenharia de software e fornecer uma base para o entendimento do restante do livro. Depois de ler este capítulo, você:

- entenderá o que é engenharia de software e por que ela é importante;
- saberá as respostas para as questões-chave que fornecem uma introdução à engenharia de software;
- entenderá questões profissionais e éticas, relevantes para os engenheiros de software.

Conteúdo

1.1 Perguntas mais freqüentes sobre engenharia de software

1.2 Responsabilidade profissional e ética

Praticamente todos os países, hoje em dia, dependem de sistemas complexos baseados em computadores. Infra-estruturas e serviços nacionais contam com sistemas baseados em computadores, e a maioria dos produtos elétricos inclui um computador e um software de controle. A manufatura e a distribuição industriais estão completamente automatizadas, assim como os sistemas financeiros. Portanto, produzir e manter o software dentro de custos adequados é essencial para o funcionamento da economia nacional e internacional.

A engenharia de software é um ramo da engenharia cujo foco é o desenvolvimento dentro de custos adequados de sistemas de software de alta qualidade. Software é abstrato e intangível. Não é limitado por materiais ou controlado por leis da física ou por processos de manufatura. De alguma maneira, isso simplifica a engenharia de software, pois não existem limitações físicas no potencial de software. Contudo, a falta de restrições naturais significa que o software pode facilmente se tornar extremamente complexo e, portanto, muito difícil de ser compreendido.

O conceito de *engenharia de software* foi inicialmente proposto em 1968, em uma conferência organizada para discutir o que foi então chamado de 'crise de software'. A crise de software resultava diretamente da introdução de novo hardware de computador baseado em circuitos integrados. Seu poder fez das aplicações de computador, consideradas até então não realizáveis, propostas viáveis. O software resultante era ordens de grandeza maior e mais complexo que sistemas anteriores de software.

A experiência inicial na construção desses sistemas mostrou que o desenvolvimento informal de software não era suficiente. Projetos importantes apresentavam, algumas vezes, anos de atraso. O software, cujo custo superava as previsões, não era confiável, era difícil de manter e seu desempenho era insatisfatório. O desenvolvimento de software estava em crise. Os custos de hardware

estavam caindo, enquanto os custos de software aumentavam rapidamente. Novas técnicas e métodos eram necessários para controlar a complexidade inerente aos grandes sistemas de software.

Essas técnicas tornaram-se parte da engenharia de software e são amplamente usadas hoje em dia. No entanto, assim como aumentou a habilidade de produzir software, cresceu também a necessidade por sistemas de software mais complexos. Novas tecnologias resultantes da convergência de computadores e sistemas de comunicação, e as complexas interfaces com o usuário, impuseram novos desafios aos engenheiros de software. Como muitas empresas ainda não aplicam as técnicas de engenharia de software de forma efetiva, muitos projetos produzem software de baixa confiabilidade, com atraso e com custo além do orçamento.

Penso que fizemos enormes progressos desde 1968 e que o desenvolvimento de engenharia de software melhorou acentuadamente nosso software. Entendemos melhor as atividades envolvidas no desenvolvimento de software. Desenvolvemos métodos eficazes de especificação, projeto e implementação de software. Novas notações e ferramentas reduzem o esforço necessário para produzir sistemas complexos e de grande porte.

Sabemos agora que não existe uma abordagem única 'ideal' para a engenharia de software. A ampla diversidade de tipos de sistemas e organizações, que usam esses sistemas, significa que precisamos de uma diversidade de abordagens para o desenvolvimento de software. No entanto, noções fundamentais de processo e de organização de sistemas constituem a base de todas essas técnicas que constituem a essência da engenharia de software.

Os engenheiros de software podem, certamente, se orgulhar de suas realizações. Sem softwares complexos, não teríamos explorado o espaço, não existiriam a Internet e as modernas telecomunicações e todos os meios de viagem seriam mais perigosos e caros. A engenharia de software tem contribuído muito e creio que, à medida que essa disciplina amadurece, sua contribuição no século XXI será ainda maior.

1.1 Perguntas mais freqüentes sobre engenharia de software

Esta seção tem como propósito responder a algumas perguntas fundamentais sobre engenharia de software e fornecer ao leitor o ponto de vista do autor sobre a disciplina. O formato utilizado nesta seção é o de lista de perguntas mais freqüentes (FAQ — Frequently Asked Questions). Essa abordagem é comumente usada nos grupos de discussão da Internet para fornecer, aos iniciantes, as respostas às perguntas mais freqüentemente levantadas. Acredito que esta é uma maneira muito eficaz de oferecer uma introdução simples à engenharia de software.

A Tabela 1.1 resume as respostas às perguntas desta seção.

1.1.1 O que é software?

Muitas pessoas associam o termo *software* aos programas de computador. Na verdade, essa é uma visão muito restritiva. Software não é apenas o programa, mas também todos os dados de documentação e configuração associados, necessários para que o programa opere corretamente. Um sistema de software consiste, geralmente, de um conjunto de programas separados; arquivos de configuração, que são utilizados para configurar esses programas; documentação do sistema, que descreve a estrutura do sistema; a documentação do usuário, que explica como usar o sistema; e sitesWeb por meio dos quais os usuários obtêm informações recentes sobre o produto.

Os engenheiros de software estão envolvidos com o desenvolvimento de produtos de software, isto é, software que pode ser vendido para um cliente. Existem dois tipos fundamentais de produtos de software:

1. *Produtos genéricos*. São sistemas do tipo *stand-alone*, produzidos por uma organização de desenvolvimento e vendidos no mercado para qualquer cliente disposto a comprá-los. Dentre os exemplos deste tipo de produtos estão os softwares para PCs, como bancos de dados, processadores de texto, pacotes gráficos e ferramentas de gerenciamento de projetos.
2. *Produtos sob encomenda (ou personalizados)*. São os sistemas encomendados por um determinado cliente. O software é desenvolvido especialmente para aquele cliente por uma empresa de software. Dentre os exemplos deste tipo de software destacam-se os sistemas de controle de dispositivos eletrônicos, desenvolvidos para apoiar um determinado processo de negócio e sistemas de controle de tráfego aéreo.

Uma diferença importante entre esses tipos de software é que, em produtos genéricos, a organização que desenvolve o software controla sua especificação. Para produtos encomendados, a especificação é normalmente desenvolvida e controlada pela organização que compra o software. Os desenvolvedores de software devem trabalhar de acordo com essa especificação.

No entanto, a linha entre esses tipos de produtos está se tornando cada vez mais tênue. Muitas empresas de software desenvolvem um produto genérico e modificam-no de acordo com as necessidades de um cliente específico. Sistemas de

Tabela 1.1 Perguntas mais freqüentes sobre engenharia de software.

Pergunta	Resposta
O que é software?	Programas de computador e documentação associada. Os produtos de software podem ser desenvolvidos para um cliente específico ou para um mercado geral.
O que é engenharia de software?	Engenharia de software é uma disciplina de engenharia relacionada a todos os aspectos de produção de software.
Qual é a diferença entre engenharia de software e ciência da computação?	A ciência da computação está relacionada com teorias e fundamentos; a engenharia de software está relacionada com a prática de desenvolvimento e entrega de software útil.
Qual é a diferença entre engenharia de software e engenharia de sistemas?	A engenharia de sistema está relacionada a todos os aspectos de desenvolvimento de sistemas baseados em computadores, incluindo hardware, software e engenharia de processo. A engenharia de software é parte desse processo.
O que é um processo de software?	Um conjunto de atividades cujo objetivo é o desenvolvimento ou a evolução de software.
O que é um modelo de processo de software?	Uma representação simplificada de um processo de software, apresentado sob perspectiva específica.
Quais são os custos da engenharia de software?	Cerca de 60 por cento dos custos são de desenvolvimento, 40 por cento são custos de teste. Para software sob encomenda, os custos de evolução freqüentemente excedem os custos de desenvolvimento.
O que são métodos de engenharia de software?	Abordagens estruturadas para desenvolvimento de software que incluem modelos de sistema, notações, regras, recomendações de projeto e guias de processo.
O que é CASE (*computer-aided software engineering*)?	Sistemas de software que têm a intenção de fornecer apoio automatizado para atividades de processo de software. Sistemas CASE são freqüentemente usados para apoio ao método.
Quais são os atributos de um bom software?	O software deve fornecer a funcionalidade e o desempenho exigidos pelo usuário e deve ser fácil de manter, confiável e usável.
Quais são os desafios-chave da engenharia de software?	Estar à altura do aumento de diversidade, demandas para redução do tempo de entrega e desenvolvimento de software digno de confiança.

planejamento de recursos empresariais (ERP — Enterprise Resource Planning), como sistemas SAP, são os melhores exemplos desse método. Nesse caso, um sistema grande e complexo é adaptado a uma empresa, incorporando informações sobre as regras e os processos de negócio, relatórios necessários etc.

1.1.2 O que é engenharia de software?

A engenharia de software é uma disciplina de engenharia relacionada com todos os aspectos da produção de software, desde os estágios iniciais de especificação do sistema até sua manutenção, depois que este entrar em operação. Nesta definição, há duas frases importantes:

1. *Disciplina de engenharia:* os engenheiros fazem as coisas funcionarem. Eles aplicam teorias, métodos e ferramentas onde for apropriado, mas eles os usam de forma seletiva e sempre procuram descobrir soluções para os problemas, mesmo quando não existem teorias e métodos aplicáveis. Os engenheiros reconhecem também que devem trabalhar sob restrições organizacionais e financeiras, e procuram soluções sem perder de vista essas restrições.
2. *Todos os aspectos da produção de software:* a engenharia de software não está relacionada apenas com os processos técnicos de desenvolvimento de software, mas também com atividades como o gerenciamento de projeto de software e o desenvolvimento de ferramentas, métodos e teorias que apóiem a produção de software.

Em geral, os engenheiros de software adotam uma abordagem sistemática e organizada em seu trabalho, que é, freqüentemente, a maneira mais eficaz de produzir software de alta qualidade. No entanto, a engenharia procura selecionar o método mais apropriado para um conjunto de circunstâncias e uma abordagem mais criativa e menos formal pode ser mais eficaz em outras circunstâncias. O desenvolvimento menos formal é particularmente apropriado para sistemas baseados na Web, que requerem uma combinação de habilidades em projeto gráfico e em software.

1.1.3 Qual é a diferença entre engenharia de software e ciência da computação?

Essencialmente, a ciência da computação diz respeito às teorias e aos métodos que constituem a base de computadores e sistemas de software, enquanto a engenharia de software se dedica aos problemas práticos da produção de software. Algum conhecimento de ciência da computação é essencial para os engenheiros de software, da mesma forma que algum conhecimento de física é essencial para engenheiros elétricos.

O ideal seria que todos os engenheiros de software se apoiassem em teorias da ciência da computação, mas na realidade isso nem sempre ocorre. Os engenheiros de software devem, com freqüência, usar métodos *ad hoc* para desenvolver software. Teorias mais elaboradas da ciência da computação nem sempre podem ser aplicadas aos problemas reais e complexos que requerem uma solução de software.

1.1.4 Qual é a diferença entre engenharia de software e engenharia de sistemas?

A engenharia de sistemas diz respeito a todos os aspectos do desenvolvimento e da evolução de sistemas complexos, nos quais o software desempenha um papel importante. A engenharia de sistemas está, portanto, relacionada ao desenvolvimento de hardware, projeto de políticas e de processos e implantação do sistema, assim como com a engenharia de software. Os engenheiros de sistemas estão envolvidos com a especificação de sistema, com a definição da sua arquitetura geral e, em seguida, com a integração de diferentes partes necessárias para criar o sistema final. Eles estão menos envolvidos com a engenharia dos componentes do sistema (como hardware, software etc.).

A engenharia de sistemas é uma disciplina mais antiga do que a engenharia de software. As pessoas têm especificado e integrado sistemas industriais complexos, como indústrias de aeronaves e químicas, por mais cem anos. No entanto, como a porcentagem de software aumentou nos sistemas, as técnicas de engenharia de software, tais como modelagem por casos de uso e gerenciamento de configuração, estão sendo usadas no processo de engenharia de sistemas. Eu explico a engenharia de sistemas no Capítulo 2.

1.1.5 O que é processo de software?

Um processo de software é um conjunto de atividades e resultados associados que produz um produto de software. Existem quatro atividades fundamentais de processo (abordadas adiante neste livro), que são comuns a todos os processos de software. São elas:

1. *Especificação de software:* clientes e engenheiros definem o software a ser produzido e as restrições para a sua operação.
2. *Desenvolvimento de software:* o software é projetado e programado.
3. *Validação de software:* na qual o software é verificado para garantir que é o que o cliente deseja.
4. *Evolução de software:* o software é modificado para se adaptar às mudanças dos requisitos do cliente e do mercado.

Diferentes tipos de sistemas necessitam de diferentes processos de desenvolvimento. Por exemplo, um software de tempo real de uma aeronave deve ser completamente especificado antes do início do desenvolvimento, enquanto nos sistemas de comércio eletrônico a especificação e o programa são, geralmente, desenvolvidos em conjunto. Conseqüentemente, essas atividades genéricas podem ser organizadas de diferentes maneiras e descritas em níveis diferentes de detalhes, para diferentes tipos de software. Porém, o uso de um processo de software inadequado pode reduzir a qualidade ou a utilidade do produto de software a ser desenvolvido e/ou aumentar os custos de desenvolvimento.

Processos de software são explicados mais detalhadamente no Capítulo 4, e o importante tópico de aprimoramento do processo de software é abordado no Capítulo 28.

1.1.6 O que é um modelo de processo de software?

Um modelo de processo de software é uma descrição simplificada desse processo de software que apresenta uma visão dele. Os modelos de processo incluem as atividades, que fazem parte do processo de software, os produtos de software e os papéis das pessoas envolvidas na engenharia de software. Alguns exemplos dos tipos de modelo de processo de software que podem ser produzidos são:

1. *Um modelo de workflow:* mostra a seqüência de atividades ao longo do processo, com suas entradas, saídas e dependências entre elas. As atividades neste modelo representam ações humanas.
2. *Um modelo de fluxo de dados ou modelo de atividade:* representa o processo como um conjunto de atividades, no qual cada atividade realiza alguma transformação de dados. Mostra como a entrada do processo, como uma especi-

ficação, por exemplo, é transformada em uma saída, como um projeto. As atividades, nesse caso, podem representar transformações realizadas por pessoas ou por computadores.

3. *Um modelo de papel/ação:* representa os papéis das pessoas envolvidas no processo de software e as atividades pelas quais são responsáveis.

A maioria dos modelos de processo de software é baseada em um dos três modelos gerais ou paradigmas de desenvolvimento de software:

1. *O modelo em cascata:* considera as atividades apresentadas anteriormente e as representa como fases separadas de processo, como especificação de requisitos, projeto de software, implementação, teste e assim por diante. Depois que cada estágio é concluído, ele é aprovado e o desenvolvimento prossegue para o estágio seguinte.
2. *Desenvolvimento iterativo:* esta abordagem intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é desenvolvido rapidamente com base em especificações muito abstratas. É então refinado com as informações do cliente, para produzir um sistema que satisfaça as necessidades deste. O sistema pode, então, ser entregue. Como alternativa, ele pode ser reimplementado, utilizando uma abordagem mais estruturada, para produzir um sistema mais robusto e mais fácil de ser mantido.
3. *Engenharia de software baseada em componentes (*CBSE – Component Based Software Engineering*):* esta técnica supõe que partes do sistema já existam. O processo de desenvolvimento concentra-se mais na integração dessas partes do que no seu desenvolvimento a partir do início. Eu explico a CBSE no Capítulo 19.

Retorno a esses modelos de processos genéricos nos capítulos 4 e 17.

1.1.7 Quais são os custos da engenharia de software?

Não existe uma resposta simples para essa pergunta, pois a distribuição do custo ao longo das diferentes atividades no processo de software depende do processo e do tipo de software que está sendo desenvolvido. Por exemplo, o software de tempo real geralmente requer validação e teste mais extensos do que sistemas baseados na Web. Contudo, cada uma dessas diferentes abordagens genéricas para o desenvolvimento de software possui um perfil diferente de distribuição de custos ao longo das atividades do processo de software. Se você supuser que o custo total de desenvolvimento de um sistema complexo de software seja de cem unidades de custo, a Figura 1.1 ilustra como essas unidades são empregadas em diferentes atividades do processo.

Na abordagem cascata, os custos de especificação, projeto, implementação e integração são medidos separadamente. Observe que a integração de sistema e teste são as atividades de desenvolvimento mais caras. Normalmente, requerem cerca de 40 por cento dos custos totais de desenvolvimento mas em alguns sistemas importantes é provavelmente de pelo menos 50 por cento desses custos.

Se o software é desenvolvido usando uma abordagem iterativa, não existe uma linha precisa entre especificação, projeto e desenvolvimento. Os custos de especificação são reduzidos, pois apenas uma especificação de alto nível é produzida antes do desenvolvimento nessa abordagem. Especificação, projeto, implementação, integração e teste são realizados paralelamente com a atividade de desenvolvimento. No entanto, você precisa ainda de uma atividade de teste de sistema independente, uma vez que a implementação inicial esteja completa.

A engenharia de software baseada em componentes tem sido aplicada intensamente apenas nos últimos tempos. Não temos ainda valores precisos dos custos de diferentes atividades de desenvolvimento de software nessa abordagem. Contudo, sabemos que os custos de desenvolvimento são menores que os custos de integração e de teste. Esses custos aumentam porque é necessário assegurar que os componentes utilizados realmente satisfazem às especificações e funcionam, conforme esperado com outros componentes.

Além dos custos de desenvolvimento, os custos também incorrem em alterações no software depois da sua liberação para uso. Os custos de evolução variam muito, dependendo do tipo de sistema. Para sistemas de software de vida longa, tais como sistemas de comando e controle, que podem ser usados por dez anos ou mais, esses custos provavelmente excedem de três a quatro vezes os custos de desenvolvimento, conforme ilustrado na Figura 1.2. Por outro lado, sistemas de pequenas empresas possuem uma vida mais curta e, conseqüentemente, custos de evolução reduzidos.

Essas distribuições de custo são válidas para software sob encomenda, conforme especificado pelo cliente e desenvolvido por um fornecedor. Para produtos de software que são vendidos (em sua maioria) para PCs, o perfil do custo é provavelmente diferente. Esses produtos são geralmente desenvolvidos a partir de um esboço de especificação, usando uma abordagem evolucionária de desenvolvimento e, portanto, os custos de especificação são relativamente baixos. No entanto, como esses produtos estão previstos para serem usados em uma faixa de diferentes configurações, eles devem ser testados intensamente. A Figura 1.2 mostra os tipos de perfil de custo esperados para esses produtos.

Figura 1.1

Distribuição de custos nas atividades de engenharia de software

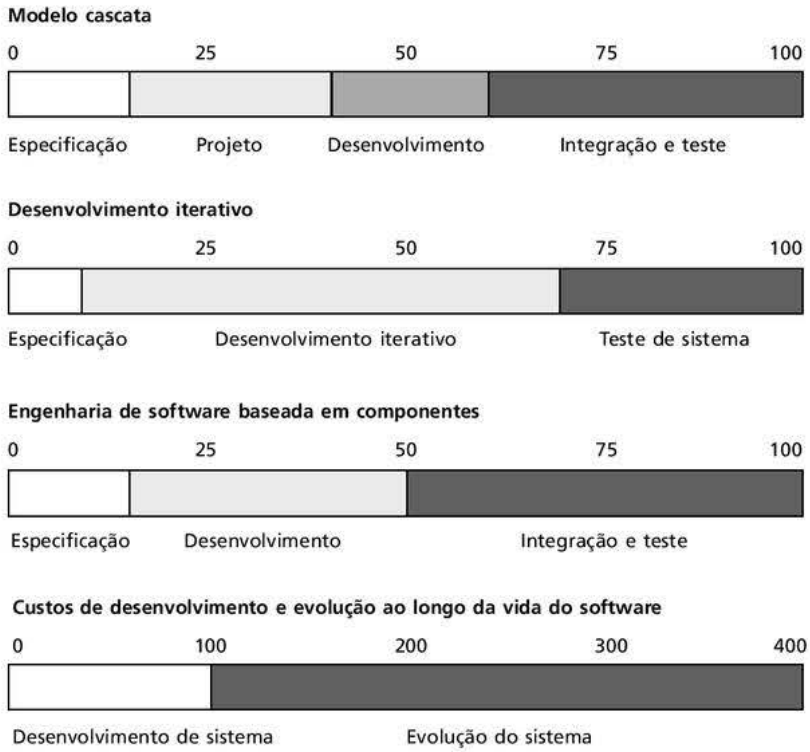
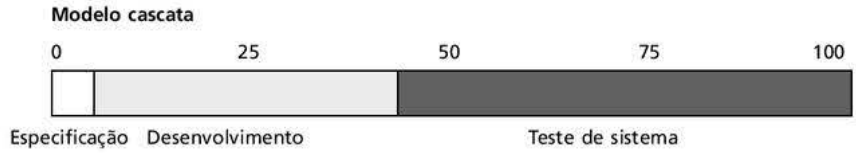


Figura 1.2

Custos de desenvolvimento do produto



Os custos de evolução para os produtos genéricos de software são particularmente difíceis de serem estimados. Em vários casos, existe uma pequena evolução formal de um produto. Uma vez que uma versão do produto foi liberada, começa-se a trabalhar na próxima release e, por razões de marketing, essa versão será apresentada como um novo (porém compatível) produto, em vez de ser apresentada como uma versão modificada do produto que o usuário já comprou. Portanto, os custos de evolução não são avaliados separadamente como no caso de software sob encomenda, mas são simplesmente custos de desenvolvimento para a próxima versão do sistema.

1.1.8 O que são métodos de engenharia de software?

Um método de engenharia de software é uma abordagem estruturada para desenvolvimento de software, cujo objetivo é facilitar a produção de software de alta qualidade dentro de custos adequados. Métodos tais como Análise Estruturada (DeMarco, 1978) e JSD (Jackson, 1983) foram desenvolvidos inicialmente na década de 1970. Esses métodos tentaram identificar os componentes funcionais básicos de um sistema; os métodos orientados a funções ainda são usados. Nas décadas de 1980 e 1990, os métodos orientados a funções foram suplementados por métodos orientados a objetos (OO), como os propostos por Booch (Booch, 1994) e Rumbaugh (Rumbaugh, et al., 1991). Essas diferentes abordagens foram agora integradas em uma abordagem única, criada de acordo com a Unified Modeling Language (UML) (Booch, et al., 1999; Rumbaugh, et al., 1999a; Rumbaugh, et al., 1999b).

Não existe um método ideal, e diferentes métodos possuem diferentes áreas onde são mais aplicáveis. Por exemplo, os métodos orientados a objetos são geralmente apropriados para sistemas interativos, mas não para sistemas com requisitos rigorosos de tempo real.

Todos os métodos estão baseados na idéia de modelos de desenvolvimento de um sistema, que pode ser representado graficamente, e na idéia de uso desses modelos como especificação e projeto de um sistema. Os métodos incluem um número de componentes diferentes (Tabela 1.2).

Tabela 1.2 Componentes do método.

Componente	Descrição	Exemplo
Descrições de modelos de sistema	Descrições de modelos de sistema que devem ser desenvolvidos e a notação usada para definir esses modelos.	Modelos de objetos, modelos de fluxo de dados, modelos de máquina de estados etc.
Regras	Restrições que são sempre aplicáveis aos modelos de sistema.	Cada entidade em um modelo de sistema deve ter um único nome.
Recomendações	Heurísticas que caracterizam uma boa prática de projeto nesse método. Seguindo essas recomendações, deve-se chegar a um modelo bem organizado do sistema.	Nenhum objeto deve ter mais que sete subobjetos associados a ele.
Guia de processo	Descrições das atividades que devem ser seguidas para desenvolver os modelos de sistema e a organização dessas atividades.	Atributos de objetos devem ser documentados antes de definir as operações associadas com um objeto.

1.1.9 O que é CASE?

O acrônimo CASE corresponde a *Computer-Aided Software Engineering*. Ele abrange uma larga faixa de diferentes tipos de programas que são usados para dar apoio às atividades do processo de software, tais como análise de requisitos, modelagem de sistema, depuração e teste. Todos os métodos vêm atualmente com uma tecnologia CASE associada, tais como editores para as notações usadas no método, módulos de análise, que verificam o modelo de análise de acordo com as regras do método, e geradores de relatório para auxiliar na criação da documentação do sistema. As ferramentas CASE podem também incluir um gerador de código que gera automaticamente o código-fonte com base no modelo do sistema, e algumas orientações sobre o processo para os engenheiros de software.

1.1.10 Quais são os atributos de um bom software?

Assim como os serviços que ele fornece, os produtos de software possuem outros atributos associados que demonstram a qualidade do software. Esses atributos não estão relacionados diretamente com o que o software faz. Em vez disso, refletem o comportamento do software, enquanto este está em execução, e a estrutura e a organização do programa-fonte bem como a documentação associada. Exemplos desses atributos (algumas vezes chamados de atributos não funcionais) são o tempo de resposta do software a uma consulta do usuário e a facilidade de entendimento do código do programa.

O conjunto específico de atributos que você pode esperar de um sistema de software depende, obviamente, da sua aplicação. Portanto, um sistema bancário deve ser seguro, um jogo interativo deve ter resposta ágil, um sistema de comutação telefônica deve ser confiável e assim por diante. Isso pode ser generalizado em um conjunto de atributos apresentado na Tabela 1.3 que, acredito, são consideradas as características essenciais de um sistema de software bem projetado.

Tabela 1.3 Atributos essenciais de um bom software.

Característica do produto	Descrição
Facilidade de manutenção	O software deve ser escrito de modo que possa evoluir para atender às necessidades de mudança dos clientes. É um atributo fundamental, pois a mudança de software é uma conseqüência inevitável de um ambiente de negócios em constante mutação.
Confiança	O nível de confiança do software tem uma série de características, incluindo confiabilidade, proteção e segurança. Um software confiável não deve causar danos físicos ou econômicos no caso de falha no sistema.
Eficiência	O software não deve desperdiçar os recursos do sistema, como memória e ciclos do processador. Portanto, a eficiência inclui tempo de resposta, tempo de processamento, utilização de memória etc.
Usabilidade	O software deve ser usável, sem esforço excessivo, pelo tipo de usuário para o qual ele foi projetado. Isso significa que ele deve apresentar uma interface com o usuário e documentação adequadas.

1.1.11 Quais são os desafios-chave da engenharia de software?

A engenharia de software no século 21 depara com três desafios-chave:

1. *O desafio da heterogeneidade:* cada vez mais, é necessário que os sistemas de software operem como sistemas distribuídos, através de redes, que incluem diferentes tipos de computadores, com diferentes tipos de sistemas de apoio. É freqüentemente necessário integrar novo software com sistemas herdados mais antigos, escritos em diferentes linguagens de programação. O desafio de heterogeneidade consiste em desenvolver técnicas para construção de software confiável que seja flexível o suficiente para adaptar-se a essa heterogeneidade.
2. *O desafio da entrega:* muitas técnicas tradicionais de engenharia de software demandam tempo. O tempo que elas necessitam é necessário para obter a qualidade do software. No entanto, o ambiente de negócios de hoje deve apresentar resposta ágil e mudar rapidamente. O software de apoio deve acompanhar a velocidade da mudança. O desafio da entrega consiste em diminuir os tempos de entrega dos sistemas grandes e complexos, sem comprometer a sua qualidade.
3. *O desafio da confiança:* como o software está entrelaçado com todos os aspectos da nossa vida, é essencial que possamos confiar nele. Isso é especialmente verdadeiro para sistemas de software remotos, acessados através de uma página Web ou interface de Web service. O desafio da confiança é desenvolver técnicas que demonstrem que o software pode ter a confiança dos seus usuários.

Naturalmente, esses desafios não são independentes. Por exemplo, pode ser necessário fazer rápidas mudanças no sistema herdado para provê-lo com uma interface de Web service. Para atender a esses desafios, necessitaremos de ferramentas e técnicas novas, bem como formas inovadoras para combinar e usar métodos existentes de engenharia de software.

1.2 Responsabilidade profissional e ética

Como outras disciplinas de engenharia, a engenharia de software está dentro de uma estrutura de condições legais e sociais que limita a liberdade dos engenheiros que, por sua vez, devem admitir que o seu trabalho implica responsabilidades mais amplas do que a aplicação de habilidades técnicas. Devem também se comportar de forma responsável ética e moralmente, para serem respeitados como profissionais.

Sem mencionar que você deve sempre defender padrões normais de honestidade e integridade. Você não deve usar sua habilidade e sua capacidade para se comportar de maneira desonesta ou de modo que trará má reputação para a profissão de engenharia de software. No entanto, existem áreas em que os padrões de um comportamento aceitável não estão limitados por leis, mas por uma tênue noção de responsabilidade profissional. Algumas delas são:

1. *Confidencialidade:* você deve respeitar a confidencialidade de seus funcionários ou clientes, independentemente de ter ou não assinado um acordo formal.
2. *Competência:* você não deve desvirtuar seu nível de competência. Você não deve conscientemente aceitar um trabalho que esteja fora da sua competência.
3. *Direitos sobre propriedade intelectual:* você deve estar ciente das leis locais que regem o uso de propriedade intelectual tais como patentes e direitos autorais. Você deve tomar cuidado para assegurar que a propriedade intelectual de funcionários e clientes seja protegida.
4. *Mau uso de computadores:* você não deve usar suas habilidades técnicas para fazer mau uso dos computadores de outras pessoas. O mau uso de computadores varia desde o relativamente trivial (execução de jogos na máquina do funcionário, por exemplo) até o extremamente sério (disseminação de vírus).

Sociedades e instituições profissionais têm um importante papel a desempenhar, definindo padrões de ética. Organizações tais como a ACM, a IEEE (Institute of Electrical and Electronic Engineers) e British Computer Society publicaram um código de conduta profissional ou código de ética. Os membros dessas organizações comprometem-se a seguir esse código quando se inscrevem como membros. Esses códigos de conduta estão geralmente relacionados com o comportamento fundamental ético.

A ACM e o IEEE têm cooperado para produzir um código comum de ética e de prática profissional. Esse código existe no formato resumido, apresentado no Quadro 1.1, e no formato mais longo (Gotterbarn, et al., 1999), que acrescenta detalhes e conteúdo à versão resumida. O fundamento desse código está resumido nos dois primeiros parágrafos da versão longa:

Os computadores desempenham um papel central e crescente no comércio, na indústria, no governo, na medicina, na educação, no entretenimento e na sociedade em geral. Os engenheiros de software são aqueles que contribuem, por participação direta ou pelo ensino, com a análise, especificação, projeto, desenvolvimento, certificação, manutenção e teste de sistemas de software. Em razão de seu papel no desenvolvimento de sistemas de software, eles têm oportunidades

Quadro 1.1 Código de Ética do ACM/IEEE (©IEEE/ACM 1999).

Código de ética e prática profissional da engenharia de software

Força-tarefa da ACM/IEEE-CS sobre éticas e práticas profissionais da engenharia de software

Preâmbulo

A versão resumida do código apresenta as aspirações em um alto nível de abstração: as cláusulas que estão incluídas na versão completa fornecem exemplos e detalhes de como essas aspirações mudam a maneira como agimos como profissionais de engenharia de software. Sem essas aspirações, os detalhes podem se tornar muito específicos e tediosos; sem os detalhes, as aspirações podem se tornar aparentemente importantes, mas vazias; juntos, aspirações e detalhes formam um código coeso.

Os engenheiros de software devem se comprometer a fazer da análise, especificação, desenvolvimento, teste e manutenção de software uma profissão benéfica e respeitada. De acordo com seu comprometimento com a saúde, segurança e bem-estar do público, os engenheiros de software devem aderir aos seguintes Oito Princípios:

1. PÚBLICO: os engenheiros de software devem agir consistentemente com o interesse público.
2. CLIENTE E EMPREGADOR: os engenheiros de software devem agir dentro dos melhores interesses de seu cliente e empregador, de forma consistente com o interesse público.
3. PRODUTO: os engenheiros de software devem assegurar que seus produtos e as modificações a eles relacionadas atendam aos mais altos padrões profissionais possíveis.
4. JULGAMENTO: os engenheiros de software devem manter a integridade e a independência em seu julgamento profissional.
5. GERENCIAMENTO: os gerentes e líderes de engenharia de software devem aceitar e promover uma abordagem ética no gerenciamento de desenvolvimento e manutenção de software.
6. PROFISSÃO: os engenheiros de software devem promover a integridade e a reputação da profissão de forma consistente com o interesse público.
7. COLEGAS: os engenheiros de software devem ser honestos e colaborativos com seus colegas.
8. INDIVÍDUO: os engenheiros de software devem participar, ao longo da vida, aprendendo, respeitando e promovendo uma abordagem ética na prática da profissão.

significativas de praticar o bem ou de causar o mal, tornarem outros capazes de praticar o bem ou causar o mal ou de influenciar outros indivíduos a praticar o bem ou causar o mal. Para garantir, tanto quanto possível, que seus esforços sejam utilizados para o bem, os engenheiros de software devem se comprometer a fazer da engenharia de software uma profissão benéfica e respeitada. De acordo com esse compromisso, os engenheiros de software devem aderir ao seguinte Código de Ética e Prática Profissional.

O Código contém oito princípios relacionados ao comportamento e às decisões tomadas pelos engenheiros de software profissionais, incluindo praticantes, educadores, gerentes, supervisores e responsáveis pela definição de políticas, assim como estagiários e estudantes da profissão. Os princípios identificam as relações de responsabilidade ética das quais indivíduos, grupos e organizações participam, assim como as principais obrigações dentro desse relacionamento. As cláusulas de cada princípio são ilustrações de algumas das obrigações incluídas nessas relações. Essas obrigações estão fundamentadas na humanidade do engenheiro de software, no cuidado especial com as pessoas afetadas pelo trabalho dos engenheiros de software e nos elementos singulares da prática da engenharia de software. O Código prescreve esses princípios como obrigações de qualquer um que diga ser, ou aspira a se tornar, um engenheiro de software.

Em qualquer situação em que pessoas diferentes têm pontos de vista e objetivos divergentes, você pode estar diante de dilemas éticos. Por exemplo, se uma pessoa discorda, em princípio, das políticas da alta gerência em sua empresa, como você deve reagir? Obviamente, isso depende dos indivíduos e da natureza da discordância. É melhor discutir um caso de acordo com seu ponto de vista dentro da organização ou demitir-se, em princípio? Se você sente que existem problemas com o projeto de software, quando você deve revelá-los à gerência? Se você discuti-los enquanto forem apenas uma suspeita, você pode causar reação exagerada para a situação; se deixar para muito depois, pode tornar-se impossível resolver as dificuldades.

Tais dilemas éticos aparecem para todos nós na vida profissional e, felizmente, na maioria dos casos, são pouco relevantes ou podem ser resolvidos sem muita dificuldade. Quando não puderem ser resolvidos, os engenheiros estarão, talvez, diante de outro problema. A ação baseada em princípios pode ser demitir-se, mas isso pode afetar outras pessoas, tais como seu cônjuge ou filhos.

Uma situação particularmente difícil para os engenheiros profissionais ocorre quando um funcionário age de maneira não ética. Digamos que uma empresa seja responsável pelo desenvolvimento de um sistema de segurança muito importante e, por causa de pressões de tempo, falsifica os registros de validação de segurança. É responsabilidade do engenheiro manter a confidencialidade, alertar o cliente ou divulgar, de alguma forma, que o sistema entregue pode ser inseguro?

O problema, no caso, é que não existem referências absolutas quando se trata de segurança. Embora o sistema não tenha sido validado de acordo com critérios predefinidos, esses critérios podem ter sido muito rigorosos. O sistema pode,

na verdade, operar com segurança ao longo de sua vida. Existe também o caso de, mesmo quando validado de modo apropriado, um sistema vir a falhar e causar um acidente. Se o problema for revelado cedo demais, pode acarretar danos para o empregador e os funcionários; se não for revelado, o problema pode resultar em danos a outras pessoas.

Você deve ter seu próprio entendimento sobre esses assuntos. A posição ética apropriada depende inteiramente do ponto de vista dos indivíduos envolvidos. Nesse caso, o potencial para o dano, a extensão do dano e as pessoas afetadas pelo dano devem influenciar a decisão. Se a situação for muito perigosa, pode ser justificável divulgá-la, pela imprensa nacional (por exemplo). Contudo, você deve sempre tentar resolver a situação, respeitando os direitos de seu empregador.

Outro assunto ético é a participação em desenvolvimento de sistemas militares e nucleares. Algumas pessoas são muito sensíveis a esses assuntos e não desejam participar de nenhum desenvolvimento relacionado a sistemas militares. Outras irão trabalhar com sistemas militares, mas não em sistemas de armamentos. Ainda outras acham que a segurança nacional está acima de qualquer princípio e não têm objeções éticas a trabalhar com sistemas de armamentos.

Nesta situação, é importante que tanto os empregadores quanto os funcionários apresentem o ponto de vista uns aos outros antecipadamente. Quando uma organização está envolvida em trabalhos militares ou nucleares, ela deve especificar que os funcionários vão ter de aceitar qualquer designação de trabalho. Igualmente, se um funcionário está sendo admitido e ele deixa claro que não quer trabalhar nesse tipo de sistema, os empregadores não devem pressioná-lo para que aceite posteriormente.

A área geral de ética e responsabilidade profissional tem recebido atenção crescente nos últimos anos. Pode ser considerada sob o ponto de vista filosófico, no qual os princípios básicos de ética são considerados e a ética em engenharia de software é discutida com referência a esses princípios. Esta é uma abordagem dada por Laudon (Laudon, 1995) e em menor extensão por Huff e Martin (Huff e Martin, 1995).

No entanto, considero que essas abordagens são muito abstratas e difíceis de serem relacionadas com as experiências do dia-a-dia. Prefiro uma abordagem mais concreta incorporada aos códigos de conduta e prática. Penso que a ética é mais bem discutida no contexto de engenharia de software, e não como um assunto independente. Neste livro, portanto, não incluo discussões abstratas sobre ética, mas, quando apropriado, incluo exemplos nos exercícios que podem servir como ponto de partida para discussão em grupo sobre tópicos de ética.

PONTOS-CHAVE

- A engenharia de software é uma disciplina de engenharia relacionada a todos os aspectos de produção de software.
- Os produtos de software consistem em programas desenvolvidos e documentação associada. Os atributos essenciais do produto são: facilidade de manutenção, confiança, eficiência e aceitação.
- O processo de software inclui todas as atividades envolvidas no desenvolvimento de software. As atividades de alto nível na especificação, desenvolvimento, validação e evolução de software são parte de todos os processos de software.
- Métodos são meios organizados de produção de software. Eles incluem sugestões sobre o processo a ser seguido, as notações a serem usadas, os modelos de sistemas a serem desenvolvidos, as regras que regem esses modelos e diretrizes do projeto.
- Ferramentas CASE são sistemas de software projetados para apoiar as atividades de rotina no processo de software, como edição de diagramas de projeto, verificação da consistência de diagramas e manutenção do controle de testes de programa realizados.
- Os engenheiros de software têm responsabilidades com a profissão de engenharia e com a sociedade. Eles não devem se preocupar apenas com questões técnicas.
- Sociedades profissionais publicam códigos de conduta que definem os padrões de comportamento esperados de seus membros.

LEITURAS SUGERIDAS

Fundamentals of software engineering. Um texto geral de engenharia de software que aborda uma perspectiva um pouco diferente do assunto deste livro. (C. Ghezi, et al., Prentice Hall, 2003.)

Software engineering: the state of the practice. Uma edição especial do IEEE Software com vários artigos que explicam as práticas atuais em engenharia de software, como ela tem mudado e a extensão em que as novas tecnologias são usadas. (IEEE Software, 20 (6), nov. 2003.)

Software engineering: an engineering approach. Um texto geral de engenharia de software que aborda o assunto de forma diferente do meu livro, mas inclui vários estudos de caso úteis. (J. F. Peters and W. Pedrycz, 2000, John Wiley & Sons.)

Professional issues in software engineering. É um excelente livro que explica os assuntos legais e profissionais, bem como a ética. Prefiro sua abordagem prática a textos mais teóricos sobre a ética. (F. Bott, et al., 3rd edition, 2000, Taylor & Francis.)

Software engineering code of ethics is approved. Um artigo que explica as bases do desenvolvimento do Código de Ética de ACM/IEEE e inclui as formas resumida e estendida do código. (Comm. ACM, D. Gotterbarn, et al., out. 1999.)

"No silver bullet: essence and accidents of software engineering". Apesar de sua idade, este artigo é uma introdução geral para os problemas de engenharia de software. A mensagem principal do artigo de que não existe uma simples resposta para os problemas de engenharia de software não mudou. (F. P. Brooks, IEEE Computer, 20 (4), abr. 1987.)

EXERCÍCIOS

1.1 Com base na distribuição de custos de software explicados na Seção 1.1.6, explique por que é apropriado considerar que o software é mais do que os programas que podem ser executados pelos usuários finais do sistema.

1.2 Quais são as diferenças entre o desenvolvimento de produto genérico de software e o desenvolvimento de software sob encomenda?

1.3 Quais são os quatro principais atributos que todos os produtos de software devem ter? Sugira quatro outros atributos que às vezes podem ser significativos.

1.4 Qual é a diferença entre um modelo de processo de software e o processo de software? Sugira dois modos pelos quais um modelo de software pode ser útil para identificar possíveis aprimoramentos no processo.

1.5 Explique por que os custos de teste de sistema são particularmente altos para os produtos genéricos de software vendidos em um mercado amplo.

1.6 Os métodos de engenharia de software tornaram-se amplamente usados somente quando a tecnologia CASE tornou-se disponível para apoiá-los. Sugira cinco tipos de apoio a métodos que podem ser fornecidos pelas ferramentas CASE.

1.7 Além dos desafios de heterogeneidade, entrega rápida e confiança, identifique outros problemas e desafios que a engenharia de software provavelmente está enfrentando no século 21.

1.8 Discuta se os engenheiros profissionais devem ter um diploma, assim como médicos e advogados.

1.9 Para cada cláusula do Código de Ética apresentado no Quadro 1.1, sugira um exemplo apropriado que ilustre a cláusula.

1.10 Para ajudar a conter o terrorismo, vários países estão planejando o desenvolvimento de sistemas de computador que monitorem 24 horas populações em grandes centros urbanos. Essa medida tem, obviamente, implicações quanto à privacidade. Discuta as questões éticas envolvidas no desenvolvimento desse tipo de sistema.

2

Sistemas sociotécnicos

Objetivos

Os objetivos deste capítulo são apresentar o conceito de um sistema sociotécnico — um sistema que inclui pessoas, software e hardware — e explicar o processo de engenharia de sistemas. Depois de ler este capítulo, você:

- conhecerá o significado de um sistema sociotécnico e compreenderá a diferença entre um sistema técnico baseado em computador e um sistema sociotécnico;
- terá sido apresentado ao conceito de propriedades emergentes de sistemas, como confiabilidade, desempenho, segurança e proteção;
- compreenderá as atividades envolvidas no processo de engenharia de sistemas;
- compreenderá por que o contexto organizacional de um sistema afeta seu projeto e seu uso;
- conhecerá o significado de um 'sistema legado' e por que esses sistemas geralmente são importantes para a operação de vários tipos de negócio.

Conteúdo

2.1 Propriedades emergentes de sistemas
2.2 Engenharia de sistemas
2.3 Organizações, pessoas e sistemas de computadores
2.4 Sistemas legados

O termo *sistema* é usado universalmente. Falamos de sistemas de computadores, sistemas operacionais, sistemas de pagamentos, sistema educacional, sistema de governo etc. Existem, obviamente, usos bastante diferentes para a palavra *sistema*, embora eles compartilhem a característica de que, de alguma maneira, o sistema é mais que simplesmente a soma de suas partes.

Sistemas muito abstratos, como sistemas de governo, estão bem distantes do escopo deste livro, portanto, darei atenção especial aos sistemas que incluem computadores e que possuem alguma finalidade específica, como permitir comunicação, dar apoio à navegação e calcular salários. Uma definição funcional útil para esses tipos de sistema é:

Um sistema é o conjunto intencional de componentes inter-relacionados que funcionam juntos para atingir certo objetivo.

Essa definição geral abrange uma vasta gama de sistemas. Por exemplo, um sistema muito simples, como uma caneta, pode incluir apenas três ou quatro componentes de hardware. Por outro lado, um sistema de controle de tráfego aéreo inclui milhares de componentes de hardware e software, além de usuários humanos que tomam decisões baseadas em informações de sistemas de computador.

Sistemas que incluem software se concentram em duas categorias:

- *Sistemas técnicos baseados em computadores* são aqueles que incluem componentes de hardware e software, mas não incluem procedimentos e processos. Podemos incluir nessa categoria de sistemas televisores, telefones celulares e a maioria dos softwares de computadores pessoais. Os indivíduos e as organizações usam sistemas técnicos para algum propósito, mas o conhecimento desse propósito não é parte do sistema. Por exemplo, o processador de texto que estou usando não está ciente de que está sendo usado para escrever um livro.
- *Sistemas sociotécnicos* incluem um ou mais sistemas técnicos, mas, decisivamente, incluem também conhecimento de como o sistema deve ser usado para alcançar um objetivo maior. Isso significa que esses sistemas têm processos operacionais definidos, incluem pessoas (os operadores) como partes inerentes do sistema, são regidos pelas políticas e regras organizacionais e podem ser afetados por restrições externas, como leis e políticas regulamentadoras nacionais. Por exemplo, este livro foi criado segundo um sistema de publicação sociotécnico que inclui vários processos e sistemas técnicos.

As características essenciais de sistemas sociotécnicos são as seguintes:

1. Eles possuem propriedades emergentes, que são propriedades do sistema *como um todo*, em vez de estarem associadas às partes individuais do sistema. As propriedades emergentes dependem tanto dos componentes do sistema como dos relacionamentos entre eles. Como isso é complexo, as propriedades emergentes podem ser avaliadas somente quando o sistema for montado.
2. Eles são freqüentemente não determinísticos, ou seja, quando apresentados a uma entrada específica, podem não produzir sempre a mesma saída. O comportamento do sistema depende dos operadores humanos, e as pessoas nem sempre reagem da mesma maneira. Além disso, o uso do sistema pode criar novos relacionamentos entre seus componentes e, portanto, mudar seu comportamento emergente.
3. A extensão na qual o sistema apóia objetivos organizacionais não depende apenas do sistema em si. Depende também da estabilidade desses objetivos, dos relacionamentos e dos conflitos entre os objetivos organizacionais e de como as pessoas interpretam esses objetivos na organização. Uma nova gerência pode reinterpretar os objetivos organizacionais para os quais um sistema foi projetado, e um sistema 'bem-sucedido' pode, então, se tornar 'malsucedido'.

Neste livro, abordo os sistemas sociotécnicos que incluem hardware e software, que definiram processos operacionais e que oferecem uma interface, implementada em software para usuários humanos. Os engenheiros de software devem ter algum conhecimento de sistemas sociotécnicos e engenharia de sistemas (White, et al., 1993; Thayer, 2002), devido à importância do software nesse tipo de sistema. Por exemplo, havia menos de 10 megabytes de software no programa espacial US Apollo que levou um homem à Lua em 1969, mas há cerca de 100 megabytes de software nos sistemas de controle da estação espacial Columbus.

Uma característica de todos os sistemas é que as propriedades e o comportamento dos componentes do sistema estão fortemente interligados. O funcionamento com sucesso de cada componente depende do funcionamento dos outros componentes. Dessa forma, o software poderá operar somente se o processador estiver operacional. O processador somente poderá realizar a computação se o sistema de software que define essas funções tiver sido instalado com sucesso.

Os sistemas geralmente são hierárquicos, portanto incluem outros sistemas. Por exemplo, um sistema policial de comando e controle deve incluir um sistema de informações geográficas para fornecer detalhes da localização de incidentes. Os sistemas incluídos são chamados de *subsistemas*. Uma característica dos subsistemas é que eles podem operar como sistemas independentes. Dessa forma, o mesmo sistema de informações geográficas pode ser usado em sistemas diferentes.

Como o software é inerentemente flexível, os problemas inesperados são freqüentemente deixados para os engenheiros de software resolverem. Digamos que um radar tenha sido instalado em um local onde ocorrem imagens fantasmas. É impraticável movê-lo para um local com menos interferência, portanto, os engenheiros de sistema precisam encontrar outra maneira de remover os fantasmas das imagens. A solução pode ser aprimorar os recursos do software de processamento de imagens para remover as imagens fantasmas. Isso pode tornar o software lento de tal modo que seu desempenho torne-se inaceitável. O problema pode, então, ser caracterizado como 'falha de software', uma vez que, de fato, foi uma falha no processo do projeto do sistema como um todo.

Essa situação, na qual os engenheiros de software enfrentam o problema de aprimorar os recursos do software, sem aumentar o custo de hardware, é muito comum. Muitas, assim ditas, falhas de software não são conseqüência de problemas

inerentes ao software; são o resultado da tentativa de alteração do software para acomodar os requisitos modificados de engenharia de sistemas. Um bom exemplo disso foi a falha do sistema de bagagens do aeroporto de Denver (Swartz, 1996), em que se esperava que o software de controle pudesse lidar com as diversas limitações do equipamento usado.

A engenharia de software é, portanto, crítica para o sucesso do desenvolvimento de sistemas sociotécnicos complexos baseados em computadores. Como engenheiro de software, você não deve simplesmente se preocupar com o software em si, mas deve ter um conhecimento mais abrangente de como esse software interage com outros sistemas de software e hardware e como se supõe que ele seja usado. Esse conhecimento ajuda a compreender os limites do software, para projetá-lo melhor e para participar como um membro atuante de um grupo de engenharia de sistemas.

2.1 Propriedades emergentes de sistemas

Os relacionamentos complexos entre os componentes de um sistema mostram que o sistema é mais do que simplesmente a soma de suas partes. Ele possui propriedades próprias do sistema como um todo. Essas *propriedades emergentes* (Checkland, 1981) não podem ser atribuídas a qualquer parte específica do sistema. Ao contrário, elas aparecem apenas após a integração dos componentes do sistema. Algumas dessas propriedades podem ser derivadas diretamente das propriedades comparáveis de subsistemas. No entanto, mais freqüentemente, elas resultam dos relacionamentos complexos de subsistemas que não podem, na prática, ser derivados de propriedades dos componentes individuais do sistema. Exemplos de algumas propriedades emergentes são mostrados na Tabela 2.1.

Há dois tipos de propriedades emergentes:

1. *Propriedades emergentes funcionais* aparecem quando todas as partes de um sistema trabalham juntas para atingir um objetivo. Por exemplo, uma bicicleta tem a propriedade funcional de ser um dispositivo de transporte, após ter sido montada com seus componentes.
2. *Propriedades emergentes não funcionais* referem-se ao comportamento do sistema em seu ambiente operacional. Exemplos de propriedades não funcionais são confiabilidade, desempenho, segurança e proteção. Elas são freqüentemente críticas para sistemas baseados em computadores, pois a falha dessas propriedades em atingir um nível mínimo definido pode tornar o sistema não utilizável. Algumas funções do sistema podem não ser necessárias a todos os usuários, tornando o sistema aceitável sem elas. No entanto, um sistema que não seja confiável ou seja muito lento provavelmente será rejeitado por todos os usuários.

Para ilustrar a complexidade das propriedades emergentes, considere a propriedade de confiabilidade do sistema. Confiabilidade é um conceito complexo que deve ser sempre considerado no nível do sistema, em vez de no nível do componente individual. Os componentes em um sistema são interdependentes, de tal modo que falhas em um componente podem se propagar através do sistema e afetar o funcionamento de outros componentes. É difícil antecipar como as conseqüências das falhas de componente se propagam pelo sistema. Conseqüentemente, não é possível fazer boas estimativas da confiabilidade global de um sistema com base em dados sobre confiabilidade dos componentes do sistema.

Tabela 2.1 Exemplos de propriedades emergentes.

Propriedade	Descrição
Volume	O volume de um sistema (espaço total ocupado) varia dependendo de como a montagem dos componentes está organizada e conectada.
Confiabilidade	A confiabilidade do sistema depende da confiabilidade dos componentes, mas interações inesperadas podem causar novos tipos de falha e, portanto, afetar a confiabilidade do sistema.
Proteção	A proteção do sistema (sua capacidade de resistir a ataques) é uma propriedade complexa que não pode ser facilmente medida. Podem ocorrer ataques não previstos pelos projetistas do sistema e, dessa maneira, as proteções internas podem ser vencidas.
Facilidade de reparos	Esta propriedade reflete a facilidade com que um problema no sistema é resolvido, após sua descoberta. Isso depende da capacidade de diagnosticar o problema, acessar os componentes defeituosos e modificar ou substituir esses componentes.
Usabilidade	Esta propriedade reflete a facilidade com que o sistema pode ser usado. Isso depende dos componentes técnicos do sistema, seus operadores e seu ambiente operacional.

Há três influências relacionadas à confiabilidade global de um sistema:

1. *Confiabilidade de hardware*. Qual é a probabilidade de falha de um componente de hardware e quanto tempo é necessário para reparar esse componente?
2. *Confiabilidade de software*. Qual é a probabilidade de um componente de software produzir uma saída incorreta? A falha de software é normalmente diferente da falha de hardware, pois o software não se desgasta. As falhas são em geral passageiras, de tal modo que o sistema continuará a funcionar depois que um resultado incorreto for produzido.
3. *Confiabilidade do operador*. Qual é a probabilidade de o operador de um sistema cometer um erro?

Todas essas influências estão fortemente relacionadas. A falha de hardware pode gerar sinais falsos, cujos valores ficarão fora da faixa de entradas esperadas pelo software. O software poderá, então, comportar-se de maneira não prevista. Os erros de operador ocorrem com mais freqüência em condições de estresse, por exemplo, quando estão ocorrendo falhas de sistema. Esses erros de operador podem, então, comprometer o hardware, causando mais falhas, e assim por diante. Portanto, uma falha inicial recuperável pode rapidamente evoluir para um problema sério que requeira o desligamento completo do sistema.

Do mesmo modo que a confiabilidade, outras propriedades, como desempenho e usabilidade, são difíceis de serem avaliadas, mas podem ser medidas depois que o sistema estiver em operação. Propriedades como segurança e proteção, no entanto, apresentam problemas diferentes. Nesse caso, você não está preocupado simplesmente com um atributo relacionado com o comportamento global do sistema, mas sim com um comportamento que o sistema *não* deveria apresentar. Um sistema protegido é aquele que não permite acesso não autorizado a seus dados, mas é claramente impossível prever todos os modos possíveis de acesso ao sistema e proibir explicitamente cada modo. Isto é, você somente saberá que um sistema está desprotegido quando alguém conseguir invadi-lo.

2.2 Engenharia de sistemas

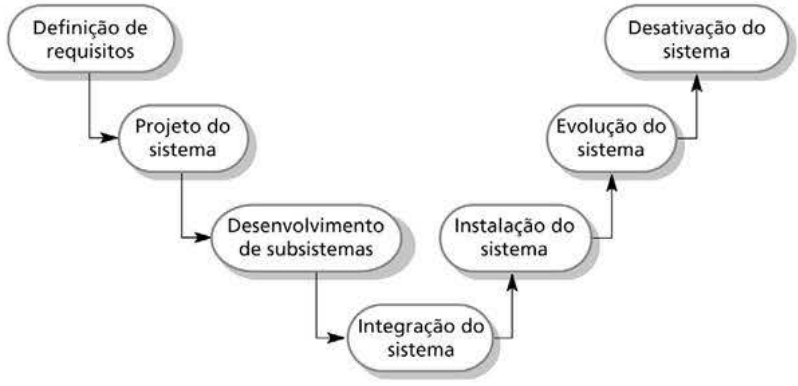
A engenharia de sistemas é a atividade de especificação, projeto, implementação, validação, implantação e manutenção de sistemas sociotécnicos. Os engenheiros de sistemas não se preocupam apenas com software, mas também com hardware e as interações do sistema com usuários e seu ambiente. Eles devem pensar sobre os serviços que o sistema fornece, as restrições sob as quais o sistema deve ser criado e operado e as maneiras de uso do sistema para atingir seu propósito. Conforme expliquei anteriormente, os engenheiros de software precisam compreender a engenharia de sistemas, pois os problemas da engenharia de software são, muitas vezes, resultado de decisões de engenharia de sistemas (Thayer, 1997; Thayer, 2002).

As fases do processo da engenharia de sistemas são mostradas na Figura 2.1. Esse processo foi uma influência importante no modelo 'cascata' do processo de software, que descrevi no Capítulo 4.

Há diferenças importantes entre o processo de engenharia de sistemas e o de desenvolvimento de software:

1. *Escopo limitado para retrabalho durante o desenvolvimento do sistema*. Após certas decisões de engenharia de sistemas, como, por exemplo, a localização das estações-base em um sistema de telefonia celular, terem sido tomadas, uma alteração é muito onerosa. O retrabalho no projeto de sistema para resolver esses problemas raramente é possível. Uma razão pela qual o software se tornou tão importante nos sistemas é que ele permite mudanças durante o desenvolvimento, em resposta a novos requisitos.
2. *Envolvimento interdisciplinar*. Muitas disciplinas de engenharia podem ser envolvidas na engenharia de sistemas. Há um escopo amplo de equívocos, pois engenheiros diferentes usam terminologia e convenções diferentes.

Figura 2.1
Processo de engenharia de sistemas



A engenharia de sistemas é uma atividade interdisciplinar, que envolve equipes formadas de vários antecedentes. As equipes de engenharia de sistemas são necessárias devido ao amplo conhecimento requisitado para considerar todas as implicações das decisões de projeto do sistema. Ilustrando isso, a Figura 2.2 mostra algumas das disciplinas que podem estar envolvidas na equipe de engenharia de sistema para um sistema de controle de tráfego aéreo que usa radares e outros sensores para determinar a posição de uma aeronave.

Para vários sistemas, existem quase infinitas possibilidades de compromissos entre os diferentes tipos de subsistemas. Diferentes disciplinas apontam para como a funcionalidade deve ser fornecida. Muitas vezes não existe uma decisão 'correta' sobre como um sistema deve ser decomposto. Ao contrário, você pode ter várias opções possíveis, mas pode não ser capaz de escolher a melhor solução técnica. Digamos que uma opção para um sistema de controle de tráfego aéreo seja construir novos radares em vez de adaptar as instalações existentes. Se os engenheiros civis envolvidos nesse processo estiverem com pouco trabalho, eles podem favorecer essa alternativa, pois ela lhes permitirá manter o emprego. Eles poderão, então, justificar essa alternativa com argumentos técnicos.

2.2.1 Definição de requisitos de sistema

As definições de requisitos de sistema especificam o que o sistema deve fazer (suas funções) e suas propriedades essenciais e desejáveis. Como na análise de requisitos de software (explicada na Parte 2), a criação das definições dos requisitos de sistema envolve consultas aos clientes e usuários finais do sistema. Essa fase de definição dos requisitos de sistema geralmente se concentra na derivação de três tipos de requisitos:

1. *Requisitos funcionais abstratos*. As funções básicas que o sistema deve fornecer são definidas no nível abstrato. A especificação de requisitos funcionais mais detalhados é realizada no nível de subsistema. Por exemplo, no sistema de controle de tráfego aéreo, um requisito funcional abstrato especifica que um banco de dados de planos de vôo deve ser usado para armazenar os planos de vôo de todas as aeronaves que penetram no espaço aéreo controlado. Contudo, você normalmente não especifica os detalhes de banco de dados, a menos que esses detalhes afetem os requisitos de outros subsistemas.
2. *Propriedades de sistema*. Essas são propriedades emergentes não funcionais de sistema tais como disponibilidade, desempenho e segurança, conforme expliquei anteriormente. Essas propriedades não funcionais de sistema afetam os requisitos de todos os subsistemas.
3. *Características que o sistema não deve apresentar*. Às vezes, especificar o que o sistema *não* deve fazer é tão importante quanto especificar o que o sistema deve fazer. Por exemplo, se você estiver especificando um sistema de controle de tráfego aéreo, poderá especificar que o sistema não apresente informações em excesso aos controladores.

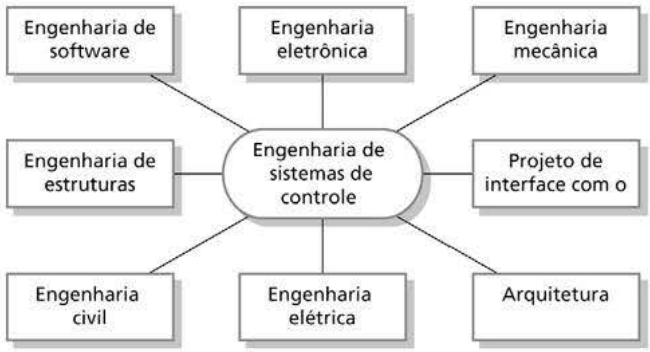
Uma importante parte da fase de definição de requisitos é estabelecer um conjunto de objetivos gerais que o sistema deve atender. Esse conjunto não deve necessariamente ser expresso em termos de funcionalidade de sistema, mas deve definir por que o sistema está sendo adquirido para um determinado ambiente.

Para ilustrar o que isso significa, digamos que você esteja especificando um sistema para um prédio de escritórios de forma a oferecer proteção contra incêndios e intrusos. Uma declaração de objetivos baseada na funcionalidade de sistema pode ser:

Fornecer um sistema de alarme contra incêndios e intrusos para o prédio que emitirá um aviso interno e externo de incêndio ou entrada não autorizada.

Esse objetivo estabelece explicitamente que há necessidade de um sistema de alarme que emita avisos para eventos não desejados. Essa declaração pode ser apropriada se você estiver substituindo um sistema de alarme existente. No entanto, uma declaração mais ampla dos objetivos pode ser:

Figura 2.2
Disciplinas envolvidas na engenharia de sistemas



Garantir que a normalidade do trabalho realizado no prédio não seja interrompida seriamente por eventos tais como incêndios e entrada de pessoas não autorizadas.

Se você estabelecer um objetivo como esse, poderá ampliar e restringir as opções de projeto. Esse objetivo permite usar, por exemplo, para proteção contra intrusos, uma tecnologia sofisticada de travamento — sem nenhum alarme interno. Também pode excluir o uso de *sprinklers* para proteção contra incêndio, pois eles podem afetar os sistemas elétricos do prédio e, então, interromper seriamente o trabalho.

Uma dificuldade fundamental em estabelecer os requisitos de sistema é que os problemas que motivaram a criação de sistemas complexos são geralmente 'problemas perversos' (Rittel e Webber, 1973). Um 'problema perverso' é um problema tão complexo e no qual existem tantas entidades relacionadas que não há uma especificação definitiva para o problema. A verdadeira natureza do problema surge somente quando uma solução é desenvolvida. Um exemplo extremo de um 'problema perverso' é um plano de terremoto. Ninguém pode predizer com precisão onde será o epicentro de um terremoto, a que horas irá ocorrer ou que tipo de efeito irá causar no ambiente local. Não podemos, portanto, especificar completamente como proceder em caso de terremoto. O problema somente poderá ser enfrentado após a sua ocorrência.

2.2.2 Projeto de sistema

O projeto de sistema (Figura 2.3) está relacionado com a maneira como a funcionalidade será fornecida pelos componentes do sistema. As atividades envolvidas nesse processo são:

1. *Agrupar requisitos*. Os requisitos são analisados e organizados em grupos relacionados. Existem, geralmente, várias opções possíveis para agrupamento e uma série de alternativas podem ser sugeridas neste estágio do processo.
2. *Identificar subsistemas*. São identificados os subsistemas que podem atender aos requisitos individual ou coletivamente. Os grupos de requisitos estão geralmente relacionados a subsistemas e, portanto, essa atividade e o agrupamento de requisitos podem ser integrados. Contudo, a identificação de subsistemas pode ser influenciada também por outros fatores organizacionais ou ambientais.
3. *Atribuir requisitos a subsistemas*. Os requisitos são atribuídos aos subsistemas. Em princípio, essa atividade deve ser direta, se o agrupamento de requisitos for usado para conduzir a identificação dos subsistemas. Na prática, nunca existe uma correspondência nítida entre os agrupamentos de requisitos e os subsistemas identificados. As limitações de subsistemas adquiridos externamente podem significar que os requisitos devem ser alterados para acomodar essas restrições.
4. *Especificar a funcionalidade de subsistemas*. As funções específicas fornecidas pelos subsistemas são definidas. Isso pode ser visto como parte da fase de projeto de sistema ou, se o subsistema for um sistema de software, parte da atividade de especificação de requisitos. Você deve também tentar identificar os relacionamentos entre os subsistemas neste estágio.
5. *Definir interfaces de subsistemas*. As interfaces fornecidas e exigidas pelos subsistemas são definidas. Após a especificação dessas interfaces, torna-se possível desenvolver subsistemas em paralelo.

Como indicam as setas bidirecionais na Figura 2.3, há uma série de realimentações e iterações de um estágio para outro nesse processo de projeto. À medida que surgem os problemas e as questões, freqüentemente é necessário retrabalho nos estágios anteriores.

Embora eu tenha separado os processos de engenharia de requisitos e projeto nesta explicação, na prática, eles estão fortemente ligados. As restrições impostas pelos sistemas existentes podem limitar as escolhas de projeto e essas escolhas podem ser especificadas nos requisitos. Pode ser necessário um projeto inicial para estruturar e organizar o processo de engenharia de requisitos. À medida que o processo de projeto prossegue, você pode descobrir problemas com os requisitos existentes e novos requisitos podem surgir. Conseqüentemente, uma maneira de pensar nesses processos ligados é como uma espiral, conforme mostrado na Figura 2.4.

Figura 2.3

Processo de projeto de sistema

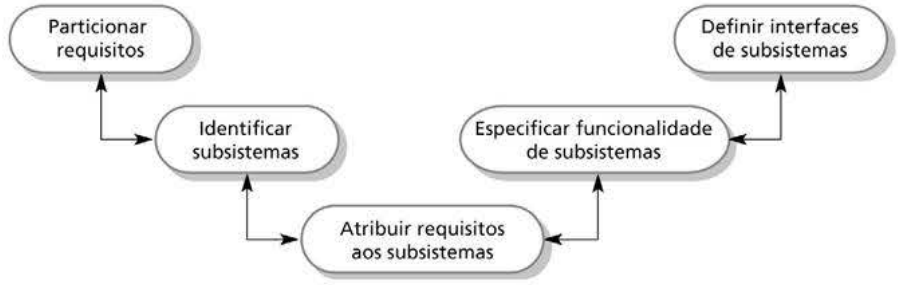
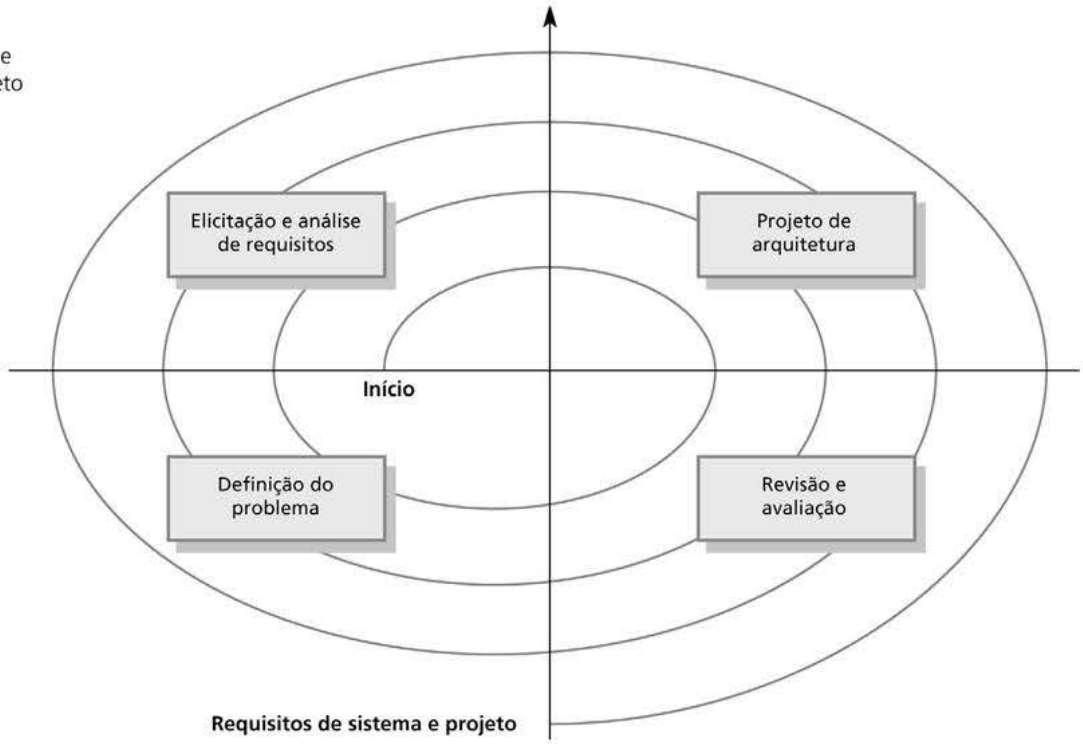


Figura 2.4
Modelo espiral de requisitos e projeto



O processo espiral reflete o fato de que os requisitos afetam as decisões de projeto e vice-versa e, assim, faz sentido interligar esses processos. Começando pelo centro, cada volta da espiral pode acrescentar detalhes aos requisitos e ao projeto. Algumas voltas podem enfocar requisitos e outras, o projeto. Às vezes, o novo conhecimento adquirido durante os processos de requisito e de projeto faz com que a própria declaração do problema precise ser alterada.

Para quase todos os sistemas, existem muitos projetos possíveis que atendem aos requisitos. Eles abrangem uma faixa de soluções que combinam hardware, software e operações humanas. A solução escolhida para o desenvolvimento posterior pode ser a solução técnica mais apropriada que atenda aos requisitos. No entanto, considerações organizacionais e políticas mais amplas podem influenciar a escolha da solução. Um cliente governamental, por exemplo, pode preferir o uso de fornecedores nacionais em lugar de estrangeiros para o seu sistema, mesmo que o produto nacional seja tecnicamente inferior. Essas influências geralmente têm efeito nas fases de revisão e avaliação no modelo espiral, no qual projetos e requisitos podem ser aceitos ou rejeitados. O processo termina quando a revisão e a avaliação mostram que os requisitos e o projeto de alto nível são suficientemente detalhados para permitir que a próxima fase do processo tenha início.

2.2.3 Modelagem de sistema

Durante as atividades de requisito e projeto, os sistemas podem ser modelados como um conjunto de componentes e relacionamentos entre esses componentes. Em geral, eles são ilustrados graficamente no modelo de arquitetura de sistema que fornece, ao leitor, uma visão geral da organização do sistema.

A arquitetura de sistema pode ser representada como um diagrama de blocos que mostra os principais subsistemas e as interconexões entre eles. Ao desenhar um diagrama de blocos, você deve representar cada subsistema por retângulos e os relacionamentos por setas que ligam esses retângulos. Os relacionamentos indicados podem incluir fluxo de dados, o relacionamento 'usa/usado por' ou algum outro tipo de relacionamento de dependência.

A Figura 2.5, por exemplo, mostra a decomposição de um sistema de alarme contra intrusos em seus principais componentes. O diagrama de blocos pode ser suplementado por breves descrições de cada subsistema, conforme mostrado na Tabela 2.2.

Nesse nível de detalhamento, o sistema é decomposto em um conjunto de subsistemas que interagem. Cada subsistema deve ser representado de forma semelhante até que o sistema esteja decomposto em componentes funcionais. Os componentes funcionais são aqueles que, quando vistos da perspectiva de um subsistema, fornecem uma função única. Por outro lado, um subsistema é geralmente multifuncional. Naturalmente, quando visto de uma outra perspectiva (digamos, do fabricante do componente), um componente funcional pode, ele próprio, ser um sistema independente.

Figura 2.5

Sistema simples de alarme contra roubo

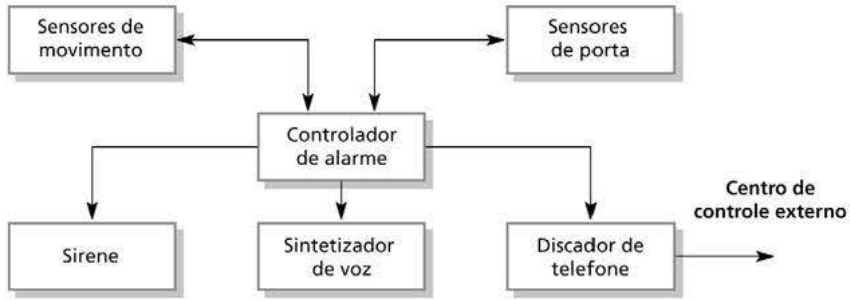


Tabela 2.2 Descrições de subsistema no sistema de alarme contra roubo.

Subsistema	Descrição
Sensores de movimento	Detecta movimento nos espaços monitorados pelo sistema
Sensores de porta	Detecta a abertura das portas externas do prédio
Controlador de alarme	Controla a operação do sistema
Sirene	Emite um alerta sonoro quando da suspeita de intrusos
Sintetizador de voz	Sintetiza uma mensagem de voz que fornece a localização do intruso suspeito
Discador de telefone	Faz chamadas externas para notificar a equipe de segurança, a polícia etc.

Historicamente, o modelo de arquitetura de sistema foi usado para identificar os componentes de hardware e software, para que pudessem ser desenvolvidos paralelamente. No entanto, essa distinção entre hardware e software tem se tornado cada vez mais irrelevante. Quase todos os componentes, atualmente, incluem alguma capacidade de computação embutida. Por exemplo, uma rede que interliga as máquinas inclui cabos físicos mais repetidores e gateways de rede. Os repetidores e gateways incluem processadores e software para direcionar esses processadores, bem como os componentes eletrônicos especializados.

No nível de arquitetura, atualmente é mais apropriado classificar os subsistemas de acordo com as suas funções, antes de tomar decisões sobre trade-offs entre hardware ou software. A decisão de fornecer uma função em hardware ou software pode ser norteada por fatores não técnicos, tais como disponibilidade de componentes comerciais ou o tempo disponível para desenvolver o componente.

Os diagramas de blocos são usados para sistemas de todos os portes. A Figura 2.6 mostra a arquitetura de um sistema muito maior para controle de tráfego aéreo. Muitos dos principais subsistemas mostrados são de grande porte. As setas que ligam esses sistemas mostram o fluxo de informações entre os subsistemas.

2.2.4 Desenvolvimento de subsistemas

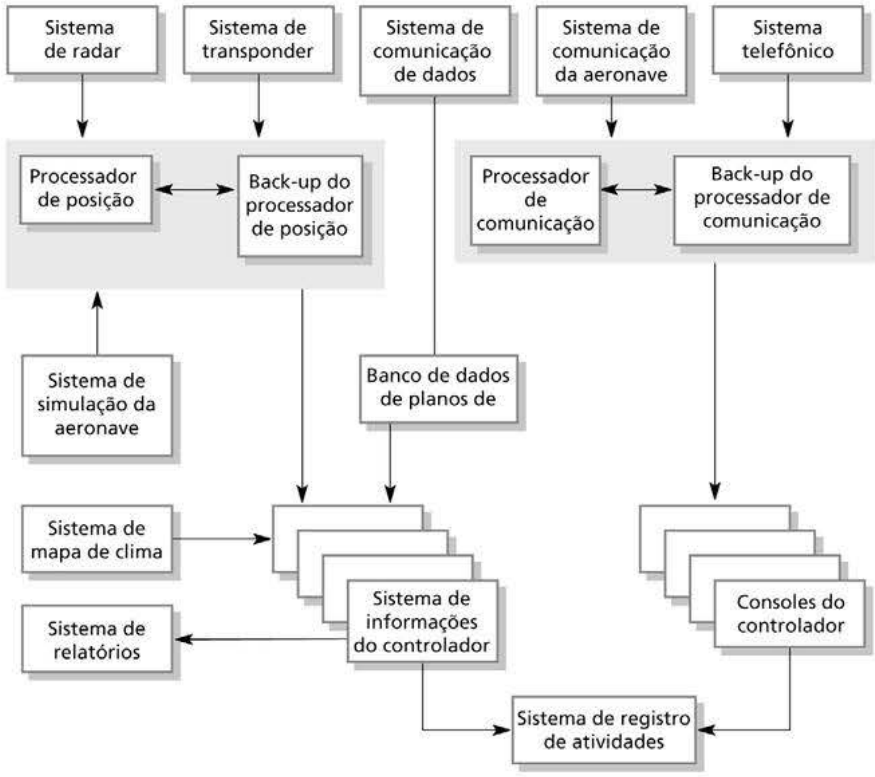
Durante o desenvolvimento de subsistemas, são implementados os subsistemas identificados durante o projeto de sistema. Isso pode envolver o início de um outro processo de engenharia de sistema para os subsistemas individuais ou, se o subsistema for software, um processo de software que envolva requisitos, projeto, implementação e teste.

Ocasionalmente, todos os subsistemas são desenvolvidos desde o início durante o processo de desenvolvimento. Contudo, normalmente alguns subsistemas são sistemas comerciais de prateleira (COTS – Commercial Off-The-Shelf) comprados para serem integrados ao sistema. Em geral é mais barato comprar produtos existentes do que desenvolver componentes específicos. Nesse estágio, a atividade de projeto pode ser retomada para acomodar um componente adquirido. Sistemas COTS podem não atender exatamente aos requisitos, mas, se produtos de prateleira estiverem disponíveis, vale a pena o esforço de repensar o projeto.

Os subsistemas são geralmente desenvolvidos de modo paralelo. Quando são encontrados problemas que ultrapassam os limites de um subsistema, uma solicitação de modificação de sistema deve ser feita. Quando os sistemas envolvem uma engenharia de hardware extensa, realizar modificações após o início da fabricação é geralmente muito oneroso. Com freqüência deve-se encontrar uma solução alternativa que compense o problema. Essa solução alternativa geralmente envolve mudanças no software, em razão de sua flexibilidade. Isso leva às mudanças nos requisitos de software e, portanto, conforme explicado no Capítulo 1, é importante projetar o software para mudanças, de tal forma que novos requisitos possam ser implementados sem excessivos custos adicionais.

Figura 2.6

Modelo de arquitetura de um sistema de controle de tráfego aéreo



2.2.5 Integração de sistemas

Durante o processo de integração de sistemas, os subsistemas desenvolvidos independentemente são agrupados para constituir um sistema completo. A integração pode ser feita por meio da abordagem *big-bang*, na qual todos os subsistemas são integrados simultaneamente. No entanto, do ponto de vista técnico e gerencial, um processo de integração incremental, no qual os subsistemas são integrados um a um, é a melhor abordagem, por dois motivos:

1. Geralmente é impossível programar o desenvolvimento de todos os subsistemas de modo que eles sejam todos concluídos ao mesmo tempo.
2. A integração incremental reduz o custo de localização de erros. Se vários subsistemas são integrados simultaneamente, um erro ocorrido durante o teste pode estar em qualquer um desses subsistemas. Quando um único subsistema é integrado com o sistema já em funcionamento, os erros ocorridos estarão provavelmente no novo subsistema integrado ou nas interações entre os subsistemas existentes e o novo subsistema.

Após a integração dos componentes, é necessário realizar um programa extensivo de testes de sistema. Esse teste deve ser voltado às interfaces entre componentes e ao comportamento do sistema como um todo.

As falhas de subsistemas em conseqüência de hipóteses inválidas sobre outros subsistemas são geralmente reveladas durante a integração do sistema. Isso pode levar a disputas entre os vários fornecedores responsáveis por diferentes subsistemas. Quando os problemas são descobertos na interação entre subsistemas, os fornecedores podem argumentar sobre qual subsistema causou a falha. As negociações a respeito de como resolver os problemas podem levar semanas ou meses.

Como cada vez mais sistemas são criados por integração de componentes COTS de hardware e software, a integração de sistemas tem se tornado cada vez mais importante. Em alguns casos, não há desenvolvimento separado de subsistemas e a integração é, essencialmente, a fase de implementação do sistema.

2.2.6 Evolução do sistema

Os sistemas grandes e complexos têm tempo de vida muito longo. Durante esse período, eles são alterados para corrigir erros nos requisitos do sistema original e para implementar novos requisitos. Os computadores do sistema

provavelmente serão substituídos por máquinas novas e mais rápidas. A organização pode se reorganizar e passar a usar o sistema de maneira diferente. O ambiente externo do sistema pode mudar e, assim, forçar mudanças no sistema.

A evolução de sistema, como a evolução do software (explicada no Capítulo 21), é inerentemente onerosa por várias razões:

1. As mudanças propostas devem ser analisadas com muito cuidado, tanto da perspectiva de negócios como do ponto de vista técnico. As mudanças precisam contribuir com os objetivos do sistema e não devem simplesmente ocorrer por motivos técnicos.
2. Como os subsistemas nunca são totalmente independentes, as mudanças em um subsistema podem afetar desfavoravelmente o desempenho e o comportamento de outros subsistemas. Portanto, mudanças posteriores nesses subsistemas podem ser necessárias.
3. Os motivos das decisões do projeto original geralmente não são registrados. Os responsáveis pela evolução do sistema precisam imaginar por que determinadas decisões de projeto foram tomadas.
4. À medida que os sistemas envelhecem, sua estrutura é corrompida por mudanças e, dessa forma, o custo de novas mudanças aumenta.

Sistemas que evoluíram ao longo do tempo são freqüentemente dependentes da tecnologia obsoleta de hardware e software. Se eles tiverem um papel importante em uma organização, são conhecidos como *sistemas legados* — sistemas que a organização gostaria de substituir, mas os riscos de introduzir um novo sistema são altos. Explico alguns tópicos relacionados a sistemas legados na Seção 2.4.

2.2.7 Desativação de sistema

A desativação de sistema significa tirar o sistema de serviço, após o término de sua vida útil operacional. Para o hardware, isso pode envolver a desmontagem e a reciclagem de materiais ou o tratamento de substâncias tóxicas. O software não tem problemas físicos de desativação, mas um determinado software pode ser incorporado em um sistema para auxiliar no processo de desativação. O software pode ser usado, por exemplo, para monitorar o estado dos componentes de hardware. Quando o sistema for desativado, os componentes que não estiverem desgastados poderão então ser identificados e reusados em outros sistemas.

Se os dados do sistema que estiver sendo desativado ainda forem valiosos para sua organização, pode ser necessário convertê-los para utilização por algum outro sistema. Isso pode envolver custos significativos, pois as estruturas de dados podem estar implicitamente definidas no próprio software. É preciso analisar o software para descobrir como os dados estão estruturados e, então, escrever um programa para reorganizar os dados nas estruturas exigidas pelo novo sistema.

2.3 Organizações, pessoas e sistemas de computadores

Os sistemas sociotécnicos são sistemas empresariais com a finalidade de auxiliar na conquista de alguma meta organizacional ou de negócios. Isso pode significar aumentar as vendas, reduzir o material usado na fabricação, coletar impostos, manter a aeronave segura etc. Como esses sistemas estão incorporados em um ambiente organizacional, a aquisição, a compra, o desenvolvimento e o uso desses sistemas são influenciados pelas políticas e procedimentos da organização e pela sua cultura de trabalho. Os usuários do sistema são pessoas influenciadas pela forma como a organização é gerenciada e pelas suas interações com outras pessoas dentro e fora da organização.

Portanto, quando estiver tentando compreender os requisitos de um sistema sociotécnico, você precisa compreender seu ambiente organizacional. Se isso não for feito, os sistemas poderão não atender às necessidades de negócios e os usuários e gerentes podem rejeitar o sistema.

Os fatores humanos e organizacionais do ambiente que afetam o projeto do sistema incluem:

1. *Mudanças de processo*. O sistema requer mudanças nos processos de trabalho no ambiente? Se sim, certamente será necessário um treinamento. Se as mudanças forem significativas ou se envolverem a perda de empregos, existe perigo de que os usuários resistirão à introdução do sistema.
2. *Mudanças de trabalho*. O sistema suplanta as habilidades dos usuários ou obriga-os a mudar a maneira como trabalham? Se sim, eles podem resistir energicamente à introdução do sistema na organização. Projetos que envolvem gerentes que precisam mudar a maneira de trabalho para se adaptarem ao sistema de computador provocam ressentimentos. Os gerentes podem sentir que seu status na organização está sendo reduzido pelo sistema.

3. *Mudanças organizacionais*. O sistema muda a estrutura política de poder em uma organização? Por exemplo, se uma organização é dependente de um sistema complexo, quem sabe como operar o sistema tem um poder político muito grande.

Esses fatores humanos, sociais e organizacionais são freqüentemente críticos para determinar se o sistema atende aos seus objetivos com sucesso ou não. Infelizmente, a previsão de seus efeitos nos sistemas é muito difícil para engenheiros que têm pouca experiência em estudos sociais ou culturais. Para auxiliar o entendimento de sistemas em organizações, várias metodologias foram desenvolvidas, tais como a *sociotécnica* de Mumford (Mumford, 1989) e a *soft systems methodology* de Checkland (Checkland e Scholes, 1990; Checkland, 1981). Tem havido também estudos sociológicos extensivos dos efeitos dos sistemas baseados em computador no trabalho (Ackroyd, et al., 1992).

Idealmente, todo o conhecimento relevante de uma organização deve ser incluído na especificação de sistema de tal forma que os projetistas de sistema possam levá-lo em consideração. Na realidade, isso é impossível. Os projetistas de sistema precisam formular hipóteses baseadas em outros sistemas semelhantes e no senso comum. Se as hipóteses não estiverem corretas, o sistema pode funcionar mal de maneiras imprevisíveis. Por exemplo, se os projetistas de um sistema não perceberem que diferentes partes de uma organização podem realmente ter objetivos conflitantes, então, qualquer sistema desenvolvido no âmbito organizacional terá, inevitavelmente, alguns usuários insatisfeitos.

2.3.1 Processos organizacionais

Na Seção 2.2, apresentei um processo de engenharia de sistema que mostrou os subprocessos envolvidos no desenvolvimento de sistema. No entanto, o processo de desenvolvimento não é o único processo envolvido na engenharia de sistemas. Ele interage com o processo de compra e com o processo de uso e operação do sistema. Isso está ilustrado na Figura 2.7.

O processo de compra está normalmente incorporado à organização que irá comprar e usar o sistema (a organização cliente). O processo de aquisição do sistema está relacionado com a tomada de decisões sobre o melhor meio de a organização adquirir um sistema e sobre os melhores fornecedores desse sistema.

Sistemas grandes e complexos geralmente são constituídos de uma combinação de componentes comerciais e daqueles especialmente construídos. Um motivo para o software ser cada vez mais incluído nos sistemas é que ele permite o uso de mais componentes existentes de hardware, no qual o software atua como uma 'cola' para fazer esses componentes de hardware trabalharem efetivamente em conjunto. A necessidade de desenvolver essa 'cola' é um dos motivos de a economia com o uso de componentes comerciais nem sempre ser tão grande quanto prevista. Os sistemas COTS serão explicados com mais detalhes no Capítulo 18.

A Figura 2.8 mostra o processo de compra tanto para sistemas existentes quanto para aqueles que precisam ser especialmente projetados. Alguns pontos importantes sobre o processo mostrado nesse diagrama são:

1. Os componentes comerciais nem sempre atendem exatamente aos requisitos, a menos que os requisitos tenham sido escritos com esses componentes em mente. Portanto, a escolha de um sistema significa que você tem que encontrar uma combinação mais próxima entre os requisitos do sistema e as características dos sistemas comerciais. Você pode, assim, ter de modificar os requisitos, e isso pode gerar efeitos que se propagam em outros subsistemas.
2. Quando um sistema é especialmente criado, a especificação de requisitos funciona como a base de um contrato para a aquisição do sistema. É, portanto, um documento tanto legal quanto técnico.
3. Depois que um fornecedor do sistema for selecionado, há um período de negociação de contrato no qual você pode negociar as mudanças adicionais dos requisitos e discutir pontos como o custo das mudanças do sistema.

As principais fases do processo de desenvolvimento de sistema já foram descritas anteriormente. Os sistemas grandes e complexos são, geralmente, desenvolvidos por uma organização diferente (o fornecedor) da organização que o está adquirindo. A razão para isso é que o ramo de atividade do comprador raramente é o desenvolvimento de sistemas e, portanto, seus funcionários não têm as habilidades necessárias para desenvolver sistemas complexos. De fato, poucas organizações possuem a capacidade para projetar, criar e testar todos os componentes de um sistema grande e complexo.

O fornecedor, que é denominado geralmente *contratado principal*, pode contratar o desenvolvimento de subsistemas diferentes de outros subcontratados. Para sistemas de grande porte, tais como sistemas de controle de tráfego aéreo, um grupo

Figura 2.7

Processos de aquisição, desenvolvimento e operação

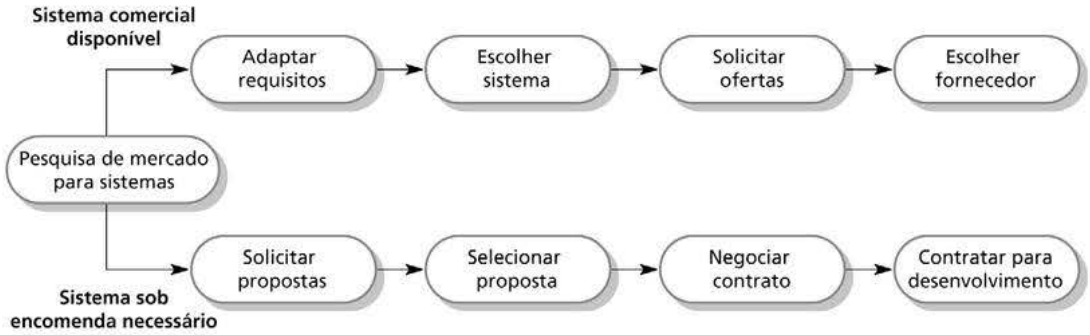


Figura 2.8 Processo de aquisição do sistema

de fornecedores pode formar um consórcio para participar de um contrato. O consórcio deve incluir todas as capacidades necessárias para esse tipo de sistema, tais como fornecedores de hardware de computador, desenvolvedores de software, fornecedores de periféricos e fornecedores de equipamento especializado, tal como radares.

O comprador negocia com o contratado em vez de com os subcontratados, de forma que exista uma única interface comprador/fornecedor. Os subcontratados projetam e criam partes do sistema para uma especificação produzida pelo contratado principal. Após a conclusão, o contratado principal integra esses componentes diferentes e libera-os para o cliente que está adquirindo o sistema. Dependendo do contrato, o comprador pode permitir que o contratado principal escolha livremente os subcontratados ou pode exigir que a escolha seja feita dentro de uma lista aprovada.

Os processos operacionais são aqueles envolvidos no uso do sistema de acordo com sua finalidade definida. Os operadores de um sistema de controle de tráfego aéreo, por exemplo, seguem um processo específico quando a aeronave entra ou sai do espaço aéreo, quando eles precisam mudar a altura ou a velocidade, quando ocorre uma emergência etc. Para novos sistemas, esses processos operacionais têm de ser definidos e documentados durante o processo de desenvolvimento. Pode ser necessário treinar os operadores e adaptar outros processos de trabalho para uso efetivo do novo sistema. Problemas não detectados podem aparecer nesse estágio, pois a especificação do sistema pode conter erros ou omissões. Embora o desempenho do sistema esteja de acordo com a especificação, suas funções podem não atender às reais necessidades operacionais. Conseqüentemente, os operadores podem não usar o sistema como os projetistas pretendiam.

O principal benefício de ter pessoas operando o sistema é que elas têm a capacidade única de responder eficientemente a situações inesperadas, mesmo que nunca tenham passado por elas. Portanto, quando as coisas não vão bem, os operadores podem contornar a situação, embora isso possa significar algumas vezes que o processo definido seja violado. Os operadores podem também usar seu conhecimento para adaptar e aprimorar os processos. Normalmente, o processo operacional real é diferente do antecipado pelos projetistas do sistema.

Isso significa que os projetistas devem desenvolver processos operacionais para que sejam flexíveis e adaptáveis. Os processos operacionais não devem ser muito restritivos, não devem exigir que operações sejam feitas em uma determinada ordem e o software de sistema não deve contar com que um processo específico seja seguido. Os operadores geralmente aprimoram o processo, pois eles sabem o que funciona ou não em uma situação real.

Um aspecto que emerge somente após a entrada do sistema em operação é o problema da operação do novo sistema com os sistemas existentes. Pode haver problemas físicos de incompatibilidade ou pode ser difícil transferir dados de um sistema para outro. Problemas mais sutis podem aparecer porque sistemas diferentes possuem diferentes interfaces com o usuário. A introdução do novo sistema pode aumentar a taxa de erros do operador em relação aos sistemas existentes, pois os usuários podem confundir os comandos da interface com o usuário.

2.4 Sistemas legados

Devido ao tempo e esforço necessários para desenvolver um sistema complexo, os sistemas de grande porte baseados em computadores têm vida longa. Sistemas militares, por exemplo, geralmente são projetados para durarem 20 anos e a maioria dos sistemas de controle de tráfego aéreo do mundo ainda conta com software e processos operacionais originalmente desenvolvidos nas décadas de 1960 e 1970. Algumas vezes é muito oneroso e muito arriscado descartar tais sistemas críticos de negócio após poucos anos de uso. Seu desenvolvimento continua ao longo da vida, com mudanças para acomodar novos requisitos, novas plataformas operacionais etc.

Os sistemas legados são sistemas sociotécnicos baseados em computadores e que foram desenvolvidos no passado, freqüentemente usando tecnologias mais antigas ou obsoletas. Esses sistemas incluem não apenas hardware e software, mas também processos e procedimentos legados — velhas formas de fazer coisas que dificilmente são mudadas porque estão baseadas em software legado. As mudanças em uma parte do sistema envolvem inevitavelmente mudanças em outros componentes.

Os sistemas legados geralmente são sistemas críticos de negócios. São mantidos porque é muito arriscado substituí-los. Por exemplo, para a maioria dos bancos, o sistema de contas de clientes foi um de seus primeiros sistemas. As políticas e os procedimentos organizacionais podem confiar nesse sistema. Se o banco fosse descartar e substituir o software de contas de clientes (executado em um hardware caro de mainframe), então haveria um risco comercial sério se o sistema substituto não funcionasse adequadamente. Além disso, os procedimentos existentes teriam de mudar, e isso poderia deixar as pessoas da organização aborrecidas e causar problemas com os auditores do banco.

A Figura 2.9 ilustra as partes lógicas de um sistema legado e seus relacionamentos:

1. *Hardware de sistema*. Em muitos casos, os sistemas legados foram escritos para hardware de mainframe que não está mais disponível; é oneroso mantê-los e eles podem não ser compatíveis com as políticas organizacionais atuais de compras de TI.
2. *Software de apoio*. O sistema legado pode contar com software de apoio, de sistema operacional e utilitários fornecidos pelo fabricante de hardware a compiladores usados no desenvolvimento do sistema. Novamente, eles podem ser obsoletos e não contarem mais com o suporte do fornecedor original.
3. *Software de aplicação*. O sistema de aplicação que fornece os serviços de negócios é geralmente composto de uma série de programas separados desenvolvidos em momentos diferentes. Às vezes, o termo *sistema legado* identifica esse sistema de software de aplicação em vez de todo o sistema.
4. *Dados de aplicação*. São os dados processados pelo sistema de aplicação. Em vários sistemas legados, um imenso volume de dados foi acumulado ao longo da vida do sistema. Esses dados podem estar inconsistentes e duplicados em vários arquivos.
5. *Processos de negócio*. São processos usados para atingir algum objetivo de negócios. Um exemplo de um processo de negócios em uma empresa de seguros seria a emissão de uma apólice de seguro; em uma empresa de manufatura, um processo de negócios seria a aceitação de um pedido de produtos e o estabelecimento do processo de manufatura associado. Os processos de negócios podem ser projetados com base em um sistema legado e serem restringidos pela funcionalidade fornecida.
6. *Políticas e regras de negócios*. São as definições de como o negócio deve ser conduzido e suas restrições. O uso do sistema de aplicação de negócios pode estar incorporado a essas políticas e regras.

Uma maneira alternativa de ver os componentes de um sistema legado é como uma série de camadas, conforme mostrado na Figura 2.10. Cada camada depende da outra imediatamente abaixo dela e das interfaces desta camada. Se as interfaces forem mantidas, então você será capaz de fazer alterações dentro de uma camada sem afetar as camadas adjacentes.

Figura 2.9

Componentes de sistema legado

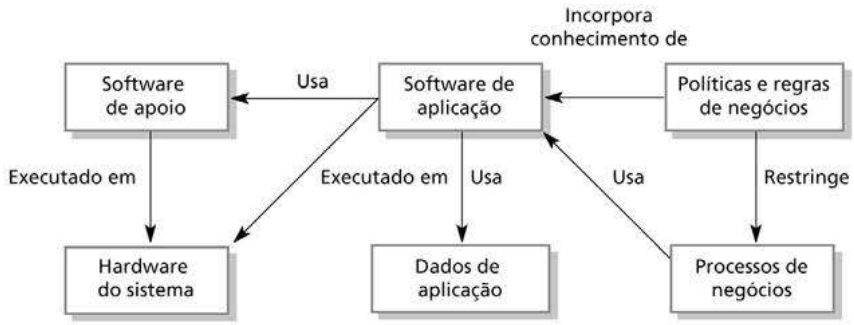


Figura 2.10

Modelo em camadas de um sistema herdado

Na prática, esse encapsulamento simples raramente funciona e as mudanças de uma camada do sistema podem exigir mudanças subseqüentes nas camadas acima e abaixo da camada alterada. As razões disso são:

1. A mudança de uma camada no sistema pode introduzir novos recursos e as camadas mais altas podem, então, ser alteradas para usufruir das vantagens desses recursos. Por exemplo, uma nova base de dados introduzida no nível da camada de apoio pode introduzir recursos de acesso aos dados por meio de um navegador Web, e os processos de negócio podem ser modificados para aproveitar esse recurso.
2. A mudança no software pode tornar o sistema lento, de tal forma que um novo hardware seja necessário para aprimorar o desempenho do sistema. O aumento de desempenho devido ao novo hardware pode, assim, significar que mudanças adicionais no software, que eram impraticáveis antes, tornem-se possíveis.
3. Geralmente é impossível manter as interfaces de hardware, especialmente se uma mudança radical for proposta para um novo tipo de hardware. Por exemplo, se uma empresa migra de hardware de mainframe para sistemas cliente–servidor (explicados no Capítulo 11), isso geralmente envolve sistemas operacionais diferentes. Mudanças maiores no software de aplicação podem, portanto, ser necessárias.

PONTOS-CHAVE

- Os sistemas sociotécnicos incluem hardware de computador, software e pessoas, e são instalados dentro de uma organização. São projetados para auxiliar a organização a atingir um grande objetivo.
- Propriedades emergentes de um sistema são características do sistema como um todo, em vez de seus componentes. São propriedades como desempenho, confiabilidade, usabilidade, segurança e proteção. O sucesso ou a falha de um sistema geralmente depende dessas propriedades.
- O processo de engenharia de sistemas inclui especificação, projeto, desenvolvimento, integração e teste. A integração de sistema, na qual os subsistemas de mais de um fornecedor devem ser colocados para trabalharem em conjunto, é particularmente crítica.
- Fatores humanos e organizacionais, tais como estrutura e política organizacionais, têm um efeito significativo sobre a operação de sistemas sociotécnicos.
- Dentro de uma organização, existem interações complexas entre os processos de aquisição, desenvolvimento e operação de sistemas.
- Um sistema legado é um sistema antigo que ainda fornece serviços essenciais de negócios.
- Sistemas legados não são apenas sistemas de software de aplicação. São sistemas sociotécnicos e, por isso, incluem processos de negócio, software de aplicação, software de apoio e hardware de sistema.

LEITURAS SUGERIDAS

"Software system engineering: a tutorial". Uma boa visão geral da engenharia de sistemas, embora Thayer enfoque exclusivamente os sistemas baseados em computadores, não abordando itens sociotécnicos. (R. H. Thayer, *IEEE computer*, abril de 2002.)

"Legacy information systems: issues and directions". Uma visão geral dos problemas de sistemas legados com um foco específico nos problemas de dados legados. (J. Bisbal, et al., *IEEE software*, setembro/outubro de 1999.)

Systems engineering: coping with complexity. Até o momento em que escrevia este material, esse ainda era o melhor livro disponível sobre engenharia de sistemas. Ele enfoca os processos de engenharia de sistemas com bons capítulos sobre requisitos, arquitetura e gerenciamento de projetos. (R. Stevens, et al., 1998, Prentice Hall.)

"Airport 95: automated baggage system". Excelente e assimilável estudo de caso sobre o que pode dar errado com o projeto de engenharia de sistemas e como o software tende a ser responsabilizado pela maioria das falhas. (*ACM software engineering notes*, 21 de março de 1996.)

EXERCÍCIOS

2.1 Explique por que outros sistemas dentro de um ambiente de sistema podem ter efeitos não previstos no funcionamento de um sistema.

2.2 Explique por que a especificação de um sistema a ser usada em serviços de emergência para gerenciamento de desastres é um 'problema perverso'.

2.3 Sugira como os sistemas de software usados em um automóvel podem auxiliar na desativação (descarte) do sistema todo.

2.4 Explique por que é importante produzir uma descrição geral da arquitetura do sistema no estágio inicial do processo de especificação do sistema.

2.5 Considere um sistema de proteção que seja uma versão estendida do sistema mostrado na Figura 2.5, cuja finalidade seja proteger contra intrusos e detectar incêndios. Ele contém sensores de fumaça, sensores de movimento, sensores de portas, câmeras de vídeo sob controle de computador localizadas em diversos locais no prédio, um console de operador no qual o status do sistema é reportado e recursos de comunicação externa para chamar os serviços apropriados, tais como polícia e bombeiros. Elabore um diagrama de blocos de um possível projeto para esse sistema.

2.6 Um sistema para alerta contra enchentes será adquirido para emitir um alerta antecipado contra possíveis perigos em locais ameaçados por enchentes. O sistema incluirá um conjunto de sensores para monitorar a taxa de variação do nível dos rios, conexões com um sistema de meteorologia que forneça a previsão de tempo, conexões com sistemas de comunicação de serviços de emergência (polícia, guarda costeira etc.), monitores de vídeo instalados em locais selecionados e uma sala de controle equipada com consoles de operador e monitores de vídeo.

Os controladores podem acessar informações de banco de dados e alternar entre displays de vídeo. O banco de dados do sistema inclui informações sobre os sensores, a posição dos locais de risco e as condições ameaçadoras nesses locais (por exemplo, maré alta, ventos vindos do sudoeste), tabelas de marés das localidades costeiras, o inventário e a localização de equipamentos de controle de enchentes, detalhes de contato de serviços de emergência, estações de rádio locais etc.

Desenhe um diagrama em blocos de uma possível arquitetura para esse sistema. Você deve identificar os principais subsistemas e as ligações entre eles.

2.7 Um sistema de museu virtual multimídia que oferece experiências virtuais da Grécia Antiga será desenvolvido para um consórcio de museus europeus. O sistema deve fornecer, aos usuários, recursos para visualizar modelos 3-D da Grécia Antiga por meio de um navegador Web padrão e deve também suportar uma experiência de imersão em realidade virtual. Que dificuldades políticas e organizacionais podem surgir quando o sistema for instalado nos museus que formam o consórcio?

2.8 Explique por que os sistemas legados podem ser críticos para a operação de um negócio.

2.9 Explique por que os sistemas legados podem causar problemas para as empresas que querem reorganizar seus processos de negócios.

2.10 Quais são os argumentos a favor e contra considerar a engenharia de sistemas como uma profissão independente, tais como a engenharia elétrica e a engenharia de software?

2.11 Você é um engenheiro envolvido no desenvolvimento de um sistema financeiro. Durante a instalação, você descobre que esse sistema tornará dispensável um número significativo de pessoas. As pessoas no ambiente não permitem seu acesso às informações essenciais para concluir a instalação do sistema. Até onde você, como engenheiro de sistema, deve se envolver nisso? É sua responsabilidade profissional concluir a instalação como contratado? Você deve simplesmente abandonar o trabalho enquanto a organização compradora esclarece o problema?

3

Sistemas críticos

Objetivos

O objetivo deste capítulo é apresentar a idéia de um sistema crítico — um sistema no qual a confiança é a propriedade mais importante. Depois de ler este capítulo, você compreenderá:

- que, em um sistema crítico, a falha pode causar graves conseqüências humanas ou econômicas;
- as quatro dimensões de confiança do sistema: disponibilidade, confiabilidade, segurança e proteção;
- que, para obter a confiança, você deve evitar erros durante o desenvolvimento de um sistema, detectar e corrigir erros quando o sistema estiver em uso e limitar os danos causados por falhas operacionais.

Conteúdo

3.1 Sistema crítico de segurança simples
3.2 Confiança no sistema
3.3 Disponibilidade e confiabilidade
3.4 Segurança
3.5 Proteção

As falhas de software são relativamente comuns. Na maioria dos casos, essas falhas causam inconveniências, mas não danos sérios no longo prazo. No entanto, em alguns sistemas, as falhas podem resultar em perdas econômicas significativas, danos físicos ou ameaças à vida humana. Esses sistemas são chamados *sistemas críticos*. São sistemas técnicos ou sociotécnicos dos quais as pessoas ou os negócios dependem. Se esses sistemas falharem ao desempenhar seus serviços conforme esperado, podem causar sérios problemas e prejuízos significativos.

Há três principais tipos de sistemas críticos:

1. *Sistemas críticos de segurança.* Sistema cuja falha pode resultar em prejuízo, perda de vida ou danos sérios ao ambiente. Um exemplo de um sistema crítico de segurança é um sistema de controle de uma fábrica de produtos químicos.
2. *Sistemas críticos de missão.* Sistema cuja falha pode resultar em problema em alguma atividade dirigida a metas. Um exemplo de um sistema crítico de missão é um sistema de navegação de uma nave espacial.
3. *Sistemas críticos de negócios.* Um sistema cuja falha pode resultar em custos muito altos para o negócio que usa esse sistema. Um exemplo de um sistema crítico de negócios é o sistema de contabilidade de clientes em um banco.

A propriedade emergente mais importante de um sistema crítico é a confiança. O termo *confiança* foi proposto por Laprie (Laprie, 1995) para abranger os atributos de sistema relacionados à disponibilidade, confiabilidade, segurança e proteção. Conforme explico na Seção 3.2, essas propriedades estão fortemente ligadas e, portanto, faz sentido adotar um termo para abranger todas elas.

Há várias razões para que a confiança seja a propriedade emergente mais importante em sistemas críticos:

1. *Sistemas não confiáveis, inseguros ou desprotegidos são freqüentemente rejeitados por seus usuários.* Se os usuários não confiam no sistema, eles se recusarão a usá-lo. Além disso, eles podem também se recusar a comprar ou a usar os produtos da mesma empresa, acreditando que esses produtos também não são confiáveis.
2. *Os custos de falha de sistema podem ser muito altos.* Para algumas aplicações, tais como um sistema de controle de reator ou um sistema de navegação de aeronaves, o custo de falha de sistema é de grandeza maior do que o custo do sistema de controle.
3. *Sistemas não confiáveis podem causar perda de informações.* Os dados são muito onerosos para serem coletados e mantidos; muitas vezes podem ser mais onerosos do que o sistema de computação que os processa. Uma grande quantidade de esforço e dinheiro pode ser despendida para duplicar dados valiosos e impedir que sejam corrompidos.

O alto custo de falha de sistemas críticos significa que métodos e técnicas confiáveis devem ser usados para seu desenvolvimento. Conseqüentemente, os sistemas críticos são, em geral, desenvolvidos usando técnicas consagradas em vez de técnicas mais recentes ainda não submetidas a ampla experiência prática. Em vez de adotarem técnicas e métodos recentes, os desenvolvedores de sistemas críticos são naturalmente conservadores. Eles preferem usar técnicas mais antigas, cujos pontos fortes e fracos são conhecidos, a técnicas recentes aparentemente melhores, mas cujos problemas no longo prazo não são conhecidos.

Técnicas onerosas de engenharia de software, não adequadas a sistemas não críticos, podem algumas vezes ser usadas no desenvolvimento de sistemas críticos. Por exemplo, métodos matemáticos formais para desenvolvimento de software (explicados no Capítulo 10) foram usados com sucesso em sistemas críticos de segurança e proteção (Hall, 1996; Hall e Chapman, 2002). Um motivo pelo qual os métodos formais são usados é que eles ajudam a reduzir a quantidade dos testes necessários. Em sistemas críticos, os custos de verificação e validação são geralmente muito altos — representam mais de 50% dos custos totais de desenvolvimento do sistema.

Embora um pequeno número de sistemas de controle possa ser totalmente automatizado, a maior parte dos sistemas críticos é constituída de sistemas sociotécnicos, nos quais as pessoas monitoram e controlam a operação dos sistemas baseados em computadores. Os custos de falha de sistemas críticos são normalmente tão altos que precisamos de pessoas no sistema que possam enfrentar situações não esperadas e contornar dificuldades.

Naturalmente, enquanto os operadores de sistema podem ajudar a superar problemas, eles podem também causar problemas, ao cometerem erros. Existem três 'componentes de sistema' nos quais as falhas de sistemas críticos podem ocorrer:

1. O hardware do sistema pode falhar por causa de erros em seu projeto, de falha dos componentes, resultante de erros de fabricação, ou de componentes que atingiram o fim de sua vida útil.
2. O software do sistema pode falhar por causa de erros em sua especificação, projeto ou implementação.
3. Os operadores humanos do sistema podem falhar em operar o sistema corretamente. Como o hardware e o software tornaram-se mais confiáveis, falhas na operação são atualmente a maior causa de falhas de sistema.

Essas falhas podem estar inter-relacionadas. A falha de um componente de hardware pode fazer com que os operadores do sistema tenham que enfrentar uma situação inesperada e uma carga adicional de trabalho. Essa situação coloca-os sob pressão — e pessoas sob pressão geralmente cometem erros. Isso pode causar uma falha de software, o que significa mais trabalho aos operadores, mais pressão e assim por diante.

Como resultado, é particularmente importante que os projetistas de sistemas críticos usem uma perspectiva holística do sistema em vez de focalizar os aspectos individuais do sistema. Se o hardware, o software e os processos operacionais forem projetados separadamente sem considerar os potenciais pontos fracos de outras partes do sistema, então é mais provável que ocorram erros nas interfaces entre as várias partes do sistema.

3.1 Sistema crítico de segurança simples

Existem vários tipos de sistemas críticos baseados em computadores, desde sistemas de controle de dispositivos e máquinas até sistemas de informações e de comércio eletrônico. Eles podem ser excelentes estudos de caso para um livro de engenharia de software, já que técnicas avançadas de engenharia de software são freqüentemente usadas em seu desenvolvi-

mento. No entanto, o entendimento desses sistemas pode ser muito difícil, pois é necessário compreender as características e as restrições do domínio de aplicação no qual eles operam.

Em decorrência, o estudo de caso de sistemas críticos que uso em vários capítulos deste livro é um sistema médico que simula o funcionamento do pâncreas (um órgão interno). Esse caso foi escolhido, pois todos nós temos algum entendimento sobre problemas médicos e é evidente por que a segurança e a confiabilidade são tão importantes para esse tipo de sistema. O sistema escolhido é destinado a auxiliar as pessoas que sofrem de diabete.

A diabete é uma condição relativamente comum quando o pâncreas humano é incapaz de produzir quantidades suficientes de um hormônio chamado *insulina*. A insulina metaboliza a glicose do sangue. O tratamento convencional de diabetes consiste na aplicação regular da insulina obtida geneticamente. Os diabéticos medem os níveis de açúcar do sangue por um medidor externo e, então, calculam a dose de insulina que devem aplicar.

O problema com este tratamento é que o nível de insulina no sangue não depende apenas do nível de glicose, mas é uma função de quando a injeção da insulina foi aplicada. Isso pode levar a níveis muito baixos de glicose no sangue (se houver muita insulina) ou a níveis muito altos de açúcar no sangue (se houver pouca insulina). O baixo nível de açúcar no sangue é, a curto prazo, uma condição mais séria, pois pode levar temporariamente a um mau funcionamento do cérebro e, por fim, à inconsciência e à morte. No longo prazo, níveis continuamente altos de açúcar no sangue podem trazer danos à vista, aos rins e problemas cardíacos.

Os avanços atuais do desenvolvimento de sensores miniaturizados determinam que, agora, é possível desenvolver sistemas automatizados de liberação de insulina. Esses sistemas monitoram os níveis de açúcar no sangue e liberam uma dose apropriada de insulina, quando necessário. Sistemas de liberação de insulina como esses já são usados em tratamento de pacientes hospitalares. No futuro, pode ser possível para vários diabéticos terem tais sistemas permanentemente ligados ao corpo.

Um sistema de liberação de insulina controlado por software poderia funcionar com o uso de um microssensor embutido no paciente para medir algum parâmetro sangüíneo proporcional ao nível de açúcar. A medida é então enviada para o controlador da bomba. Esse controlador calcula o nível de açúcar e a quantidade de insulina necessária. Em seguida, envia os sinais para uma bomba miniaturizada para liberação da insulina através de uma agulha permanentemente conectada. A Figura 3.1 mostra os componentes e a organização da bomba de insulina.

A Figura 3.2 é um modelo de fluxo de dados que ilustra como uma entrada de nível de açúcar no sangue é transformada em uma seqüência de comandos de controle da bomba.

Figura 3.1

Estrutura de bomba de insulina

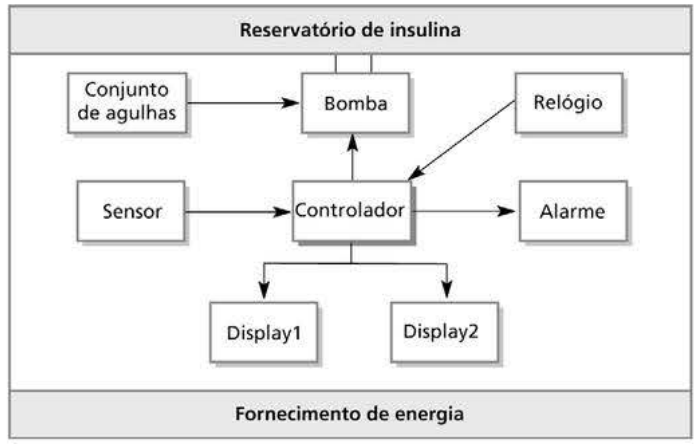
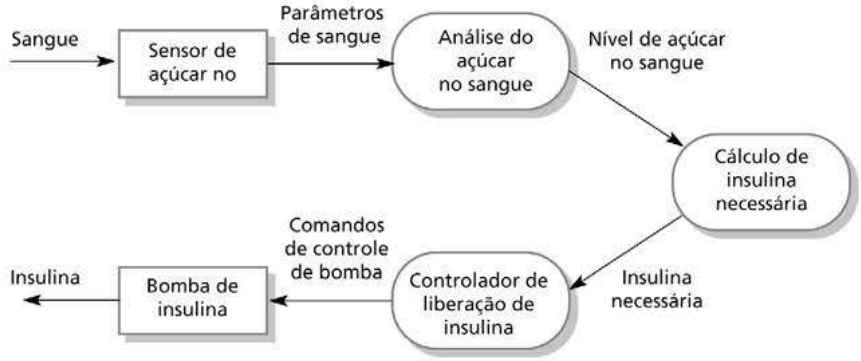


Figura 3.2

Modelo de fluxo de dados de bomba de insulina



Há dois requisitos de confiança de alto nível para esse sistema de bomba de insulina:

1. O sistema deve estar disponível para liberar a insulina quando solicitado.
2. O sistema deve funcionar de forma confiável e liberar a quantidade correta de insulina para neutralizar o nível atual de açúcar no sangue.

A falha do sistema poderia, em princípio, causar doses excessivas de insulina a serem liberadas e isso poderia ameaçar a vida do usuário. É muito importante que não ocorram overdoses de insulina.

3.2 Confiança no sistema

Todos nós estamos familiarizados com o problema da falha de sistema de computador. Sem uma razão óbvia, os sistemas de computador às vezes 'caem' e falham em fornecer os serviços solicitados. Os programas executados nesses sistemas podem não operar conforme esperado e, de vez em quando, podem corromper os dados que estão sendo gerenciados. Nós aprendemos a conviver com essas falhas e poucos confiam completamente nos computadores pessoais que normalmente usamos.

A confiança em um sistema de computador é uma propriedade do sistema cuja contrapartida é o merecimento a essa confiança. Essa propriedade significa essencialmente o grau de confiança dos usuários em que o sistema irá operar conforme sua expectativa e que não irá 'falhar' durante o uso normal. Ela não pode ser expressa numericamente, mas os termos tais como *não confiável*, *muito confiável* e *ultraconfiável* são usados para expressar o nível de confiança que podemos ter em um sistema.

Confiança e utilidade não são, obviamente, a mesma coisa. Não acho que o processador de texto que usei para escrever este livro seja um sistema muito confiável, mas é muito útil. No entanto, para refletir a minha falta de confiança no sistema, salvo freqüentemente meu trabalho e mantenho diversas cópias de back-up. Compenso a falta de confiança no sistema com ações que limitam os danos que podem ser causados caso o sistema venha a falhar.

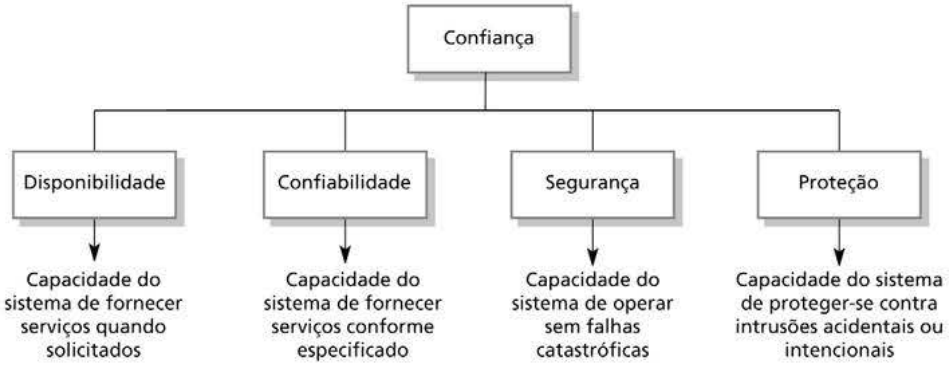
Há quatro dimensões de confiança, conforme mostrado na Figura 3.3:

1. *Disponibilidade.* Informalmente, a disponibilidade de um sistema é a probabilidade de que ele esteja pronto e em execução, capaz de fornecer serviços úteis a qualquer instante.
2. *Confiabilidade.* Informalmente, a confiabilidade de um sistema é a probabilidade, em um dado período de tempo, de que o sistema forneça corretamente os serviços, conforme esperado pelo usuário.
3. *Segurança.* Informalmente, a segurança de um sistema é um julgamento da probabilidade de que um sistema cause danos para pessoas ou para o ambiente.
4. *Proteção.* Informalmente, a proteção do sistema é um julgamento da probabilidade de que um sistema possa resistir a intrusões acidentais ou intencionais.

Essas propriedades são complexas e podem ser decompostas em uma série de outras propriedades mais simples. Por exemplo, *proteção* inclui *integridade* (garantia de que os programas e os dados do sistema não sejam danificados) e *confidencialidade* (garantia de que as informações possam ser acessadas apenas pelas pessoas autorizadas). *Confiabilidade* inclui *correção* (garantia de que os serviços do sistema estejam conforme especificados), *precisão* (garantia de que a informação seja liberada em um nível apropriado de detalhes) e *em tempo certo* (garantia de que as informações sejam liberadas quando solicitadas).

As propriedades de confiança de disponibilidade, proteção, confiabilidade e segurança estão todas inter-relacionadas. A segurança do sistema geralmente depende do fato de estar disponível e da confiabilidade da operação. Um sistema pode se

Figura 3.3
Dimensões de confiança



tornar não confiável, pois seus dados foram corrompidos por um intruso. Ataques do tipo negação de serviços a um sistema têm a intenção de comprometer a sua disponibilidade. Se um sistema considerado seguro for infectado por um vírus, a operação segura não poderá mais ser presumida. Foi por causa dessas ligações fortes que a noção de confiança do sistema como uma propriedade abrangente foi introduzida.

Assim como as quatro dimensões principais, outras propriedades de sistema podem também ser consideradas sob o aspecto da confiança:

1. *Facilidade de reparos.* As falhas de sistemas são inevitáveis, mas a interrupção causada por falhas pode ser minimizada se o sistema puder ser reparado rapidamente. Para que isso aconteça, deve ser possível diagnosticar o problema, acessar o componente que falhou e fazer alterações para consertá-lo. A facilidade de reparos de software é aprimorada quando a organização que usa o sistema tem acesso ao código-fonte e há pessoas com habilidade para fazer as alterações. Infelizmente, isso tem se tornado cada vez menos comum à medida que avançamos em direção ao desenvolvimento de sistemas por meio de componentes de terceiros do tipo caixa-preta (veja o Capítulo 19).
2. *Facilidade de manutenção.* À medida que os sistemas são usados, novos requisitos surgem. É importante manter a utilidade de um sistema, alterando-o para acomodar esses novos requisitos. Um software fácil de manter é aquele que pode ser adaptado com baixo custo para absorver os novos requisitos e no qual existe uma baixa probabilidade de que as mudanças introduzam novos erros no sistema.
3. *Capacidade de sobrevivência.* Um atributo muito importante para sistemas baseados em Internet é a capacidade de sobrevivência, que está estreitamente relacionada à proteção e à disponibilidade (Ellison, et al., 1999). A capacidade de sobrevivência é a capacidade de o sistema continuar a fornecer o serviço quando estiver sob ataque e, potencialmente, enquanto parte do sistema estiver desativada. O trabalho com capacidade de sobrevivência enfoca a identificação de componentes principais do sistema e a garantia de que eles podem fornecer um mínimo de serviço. Três estratégias são usadas para aprimorar a capacidade de sobrevivência: resistência a ataques, reconhecimento de ataque e recuperação de danos causados por um ataque (Ellison, et al., 1999; Ellison, et al., 2002).
4. *Tolerância a erros.* Esta propriedade pode ser considerada como parte da usabilidade (explicada no Capítulo 16) e reflete até onde o sistema foi projetado para evitar e tolerar erros de entrada do usuário. Quando ocorrem erros de usuário, o sistema deve, sempre que possível, detectá-los e corrigi-los ou solicitar uma nova entrada de dados ao usuário.

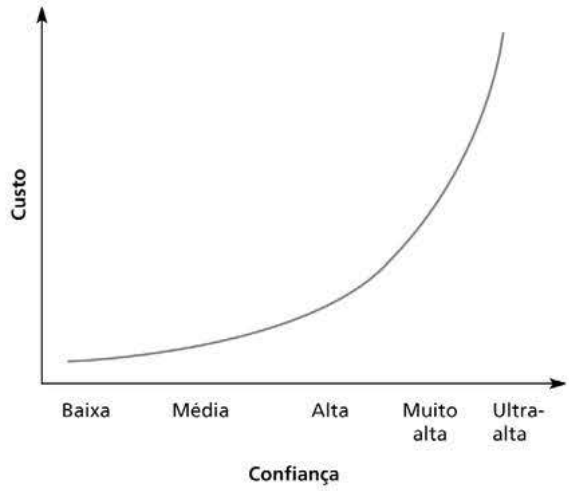
Como disponibilidade, confiabilidade, segurança e proteção são propriedades fundamentais de confiança, concentro-me neles neste capítulo e nos capítulos posteriores que cobrem especificação de sistemas críticos (Capítulo 9), desenvolvimento de sistemas críticos (Capítulo 20) e validação de sistemas críticos (Capítulo 24).

Naturalmente, as propriedades de confiança não são todas aplicáveis a todos os sistemas. Para o sistema de bomba de insulina, apresentado na Seção 3.1, as propriedades mais importantes são disponibilidade (deve funcionar quando solicitado), confiabilidade (deve liberar a dose correta de insulina) e segurança (nunca deve liberar uma dose perigosa de insulina). A proteção, neste caso, é menos provável de ser considerada, pois a bomba não manterá informações confidenciais e não estará ligada em rede, portanto, não pode ser maliciosamente atacada.

Os projetistas devem normalmente fazer um balanço entre o desempenho e a confiança do sistema. Em geral, altos níveis de confiança podem ser somente obtidos à custa do desempenho do sistema. O software confiável inclui código extra, freqüentemente redundante, para desempenhar a verificação necessária de estados excepcionais do sistema e para se recuperar

Figura 3.4

Curva custo/confiança



das falhas. Isso reduz o desempenho do sistema e aumenta a quantidade de armazenamento exigida pelo software. O custo de desenvolvimento do sistema também aumenta significativamente.

Devido aos custos adicionais de projeto, implementação e validação, o aumento da confiança de um sistema pode aumentar significativamente os custos do desenvolvimento. Os custos de validação, em particular, são altos para sistemas críticos. Além de validar o sistema quanto ao atendimento dos requisitos, o processo de validação pode ter de provar a um órgão regulamentador externo, como a Federal Aviation Authority, que o sistema é confiável.

A Figura 3.4 mostra o relacionamento entre os custos e os aprimoramentos incrementais de confiança. Quanto maior a confiança necessária, mais você precisará gastar em testes para verificar se o nível foi atingido. Devido à natureza exponencial dessa curva de custo/de confiança, não é possível demonstrar que um sistema é cem por cento confiável, pois os custos de garantia de confiança seriam infinitos.

3.3 Disponibilidade e confiabilidade

A disponibilidade e a confiabilidade são propriedades fortemente relacionadas que podem ser expressas como probabilidades numéricas. A confiabilidade de um sistema é a probabilidade de que os serviços serão fornecidos corretamente, conforme especificado. A disponibilidade de um sistema é a probabilidade de que o sistema estará pronto e funcionando para fornecer esses serviços aos usuários quando eles solicitarem.

Embora essas propriedades estejam fortemente relacionadas, você não pode presumir que sistemas confiáveis serão sempre disponíveis e vice-versa. Por exemplo, alguns sistemas podem ter um requisito de alta disponibilidade, mas um requisito de confiabilidade muito mais baixo. Se os usuários esperam um serviço contínuo, então os requisitos de disponibilidade serão altos. Mas, se as conseqüências de uma falha forem mínimas e o sistema puder se recuperar rapidamente dessa falha, então o mesmo sistema pode ter requisitos de confiabilidade baixos.

Um exemplo de sistema no qual a disponibilidade é mais importante que a confiabilidade é o sistema de comutação de telefones. Os usuários esperam o tom de discar assim que levantam o fone; portanto, o sistema tem requisitos de alta disponibilidade. No entanto, se uma falha no sistema causar um problema na conexão, esta será, geralmente, recuperada. Os comutadores de telefones incluem recursos de reparo que podem reiniciar o sistema e tentar novamente a conexão. Isso pode ser feito rapidamente e o usuário do telefone pode até mesmo deixar de perceber a ocorrência da falha. Portanto, a disponibilidade, mais do que a confiabilidade, é um requisito principal de confiança para estes sistemas.

Uma diferença adicional entre essas características é que a disponibilidade não depende simplesmente do sistema em si, mas também do tempo necessário para reparar os defeitos que tornam o sistema indisponível. Portanto, se o sistema A falhar uma vez por ano e o sistema B falhar uma vez por mês, então A é obviamente mais confiável que B. No entanto, suponha que A leve três dias para ser reiniciado após a falha, enquanto B leva 10 minutos. A disponibilidade do sistema B durante um ano (120 minutos de interrupção) é muito melhor que a do sistema A (4.320 minutos de interrupção).

A confiabilidade e a disponibilidade do sistema podem ser definidas mais precisamente conforme segue:

1. *Confiabilidade.* Probabilidade de operação livre de falhas durante um período especificado, em um dado ambiente, para um objetivo específico.
2. *Disponibilidade.* Probabilidade de que um sistema, em determinado instante, estará operacional e será capaz de fornecer os serviços solicitados.

Um dos problemas práticos no desenvolvimento de sistemas confiáveis é que as noções intuitivas de confiabilidade e disponibilidade são, algumas vezes, mais amplas que essas limitadas definições. A definição de *confiabilidade* estabelece que o ambiente no qual o sistema é usado e o objetivo para o qual ele é usado devem ser levados em consideração. Se você medir a confiabilidade do sistema em um ambiente, não será possível supor que ela será a mesma em um outro ambiente onde o sistema é usado de maneira diferente.

Por exemplo, digamos que você mediu a confiabilidade de um processador de texto em um ambiente de escritório, onde a maioria dos usuários não está interessada na operação do sistema. Eles seguem as instruções para seu uso e não tentam fazer experiências com o sistema. Se você medir a confiabilidade do mesmo sistema em um ambiente universitário, então a confiabilidade pode ser completamente diferente. Nesse caso, os estudantes podem explorar os limites do sistema e usar o sistema de maneiras inesperadas. Isso pode resultar em falhas que não ocorreriam no ambiente de escritório mais restritivo.

As percepções humanas e os padrões de uso também são significativos. Por exemplo, digamos que um automóvel tenha um defeito no sistema do limpador de pára-brisa que resulta em falhas intermitentes dos limpadores durante chuva forte. A confiabilidade desse sistema, percebida pelo motorista, depende de onde ele mora e usa o automóvel. O motorista de Seattle (clima úmido) será provavelmente mais afetado por essa falha do que o motorista de Las Vegas (clima seco). A percepção do motorista de Seattle será que o sistema não é confiável, enquanto o motorista de Las Vegas pode nunca perceber o problema.

Uma dificuldade adicional com essas definições é que elas não levam em consideração a severidade da falha ou as conseqüências da não-disponibilidade. As pessoas estão, naturalmente, mais preocupadas com as falhas de sistema que têm conseqüências sérias e sua percepção sobre a confiabilidade do sistema é influenciada por isso. Por exemplo, digamos que uma falha de iniciação no software de gerenciamento do motor causa uma parada imediata após a partida, mas opera corretamente após nova partida que corrige o problema da iniciação. Isso não afeta a operação normal do automóvel e muitos motoristas não acharão que é necessário um conserto. Contrariamente, a maioria dos motoristas achará que um motor que pára enquanto eles estão dirigindo em alta velocidade, uma vez por mês (por exemplo), não é confiável nem seguro, e deve ser consertado.

Uma definição rigorosa de confiabilidade relaciona a implementação do sistema com sua especificação. Isto é, o sistema está se comportando confiavelmente se seu comportamento for consistente com o definido na especificação. No entanto, uma causa comum da falta de confiabilidade percebida é que a especificação do sistema não atende às expectativas dos seus usuários. Infelizmente, muitas especificações são incompletas e incorretas, e a interpretação de como o sistema deve se comportar é deixada para os engenheiros de software. Como eles não são especialistas do domínio, não podem, portanto, implementar o comportamento que o usuário espera.

A confiabilidade e a disponibilidade são comprometidas pelas falhas de sistema. Essas falhas dizem respeito ao fornecimento de um serviço, à liberação de um serviço conforme especificado ou à liberação de um serviço de tal maneira que seja inseguro ou desprotegido. Algumas dessas falhas ocorrem como conseqüência de erros de especificação ou falhas em sistemas associados, tal como um sistema de telecomunicação. No entanto, muitas falhas são causadas pelo comportamento errôneo de sistema que derivam de defeitos no sistema. Quando se discute a confiabilidade, é proveitoso distinguir os termos *defeito*, *erro* e *falha*. Defini esses termos na Tabela 3.1.

Os erros humanos não levam inevitavelmente a falhas de sistema. Os defeitos introduzidos podem estar em partes do sistema que nunca foram usadas. Os defeitos não necessariamente resultam em erros de sistema, pois o estado defeituoso pode ser transitório e corrigido antes que um comportamento errôneo ocorra. Os erros de sistema podem não resultar em falhas, pois o comportamento pode também ser transitório e não apresentar efeitos observáveis ou o sistema pode incluir proteção que garanta que o comportamento errôneo seja descoberto e corrigido antes que os serviços do sistema sejam afetados.

Essa distinção entre os termos mostrada na Tabela 3.1 ajuda a identificar três abordagens complementares usadas para aprimorar a confiabilidade de um sistema:

1. *Prevenção de defeitos.* Técnicas de desenvolvimento são usadas para minimizar a possibilidade de erros e/ou detectá-los antes que causem defeitos no sistema. Exemplos de tais técnicas incluem evitar linguagem de programação propensas a erros, tais como ponteiros, e o uso de análise estática para detectar anomalias de programa.
2. *Detecção e remoção de defeitos.* O uso de técnicas de verificação e validação que aumenta as chances de detecção e remoção de defeitos antes que o sistema seja usado. O teste e a depuração sistemáticos é um exemplo de técnica de detecção de defeitos.
3. *Tolerância a defeitos.* Técnicas que garantem que defeitos em um sistema não resultem em erros de sistema ou que garantem que erros de sistema não resultem em falhas de sistema. A incorporação de recursos de autoverificação e o uso de módulos de sistema redundantes são exemplos de técnicas de tolerância a defeitos.

Abordo o desenvolvimento de sistemas tolerantes a defeitos no Capítulo 20, no qual também explico algumas técnicas para prevenção de defeitos. Explico abordagens baseadas em processos para prevenção de defeitos no Capítulo 27 e detecção de defeitos nos capítulos 22 e 23.

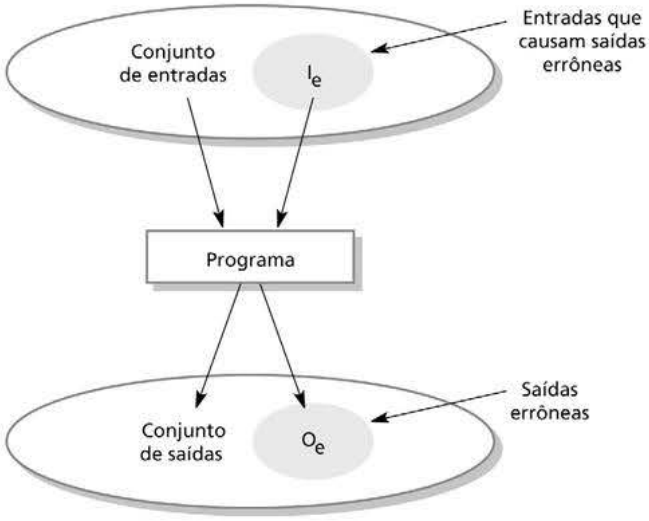
Os defeitos causam falhas de software quando o código defeituoso é executado com um conjunto de entradas que expõem o defeito de software. O código funciona de forma apropriada com a maioria das entradas. A Figura 3.5, derivada

Tabela 3.1 Terminologia de confiabilidade.

Termo	Descrição
Falha de sistema	Um evento que ocorre em algum momento, quando o sistema não fornece um serviço conforme esperado por seus usuários.
Erro de sistema	Um estado errôneo de sistema que pode levá-lo a um comportamento inesperado pelos seus usuários.
Defeito de sistema	Uma característica do sistema de software que pode levar a um erro de sistema. Por exemplo, a falha em iniciar uma variável pode levar a um valor errado quando esta for usada.
Erro humano ou engano	Comportamento humano que resulta na introdução de defeitos em um sistema.

Figura 3.5

Sistema de mapeamento de entrada/saída



de Littlewood (Littlewood, 1990), mostra um sistema de software que mapeia um conjunto de entradas para um conjunto de saídas. Por exemplo, dada uma entrada e um URL, um navegador Web produz uma saída que é a apresentação da página Web solicitada.

Algumas dessas entradas ou combinações de entradas, mostradas na elipse sombreada na Figura 3.5, causam a geração de saídas errôneas. A confiabilidade do software está relacionada com a probabilidade de que, em uma determinada execução do programa, a entrada do sistema seja parte do conjunto de entradas, o que causa a ocorrência de uma saída errônea. Se uma entrada que causa uma saída errônea estiver associada a uma parte do programa freqüentemente utilizada, as falhas serão freqüentes. Contudo, se estiver associada a um código raramente usado, os usuários dificilmente perceberão as falhas.

Cada usuário utiliza o sistema de formas diferentes. Defeitos que afetam a confiabilidade do sistema para um usuário podem nunca ser revelados sob um modo de trabalho de outra pessoa (Figura 3.6). Na Figura 3.6, o conjunto de entradas errôneas corresponde à elipse sombreada da Figura 3.5. O conjunto de entradas produzidas pelo Usuário 2 tem intersecção com o conjunto de entradas errôneas. O Usuário 2 experimentará, portanto, algumas falhas de sistema. O Usuário 1 e o Usuário 3, no entanto, nunca usam as entradas do conjunto de entradas errôneas. Para eles, o software será sempre confiável.

A confiabilidade global de um programa, portanto, depende principalmente do número de entradas que causam saídas errôneas durante o uso normal do sistema pela maioria dos usuários. Os defeitos de software que ocorrem somente em situações excepcionais têm pouco efeito sobre a confiabilidade do sistema. A correção dos defeitos de software das partes do sistema raramente usadas faz pouca diferença para a confiabilidade do ponto de vista dos usuários do sistema. Mills et al. (Mills, et al., 1987) observaram que, em seu software, a remoção de 60% dos erros conhecidos levou à melhoria de apenas 3% de confiabilidade. Adams (Adams, 1984), em um estudo de produtos de software da IBM, notou que muitos defeitos nos produtos iriam provavelmente causar falhas depois de centenas ou milhares de meses de uso do produto.

Os usuários de um sistema sociotécnico podem se adaptar ao software com defeitos conhecidos e compartilhar as informações de como contornar esses problemas. Além disso, usuários experientes freqüentemente contornam os defeitos de software conhecidos que causam falhas. Eles evitam deliberadamente o uso dos recursos de sistema que podem causar problemas. Por exemplo, eu evito certos recursos, como a numeração automática do sistema de processador de texto usado

Figura 3.6

Padrões de uso de software

para escrever este livro. O reparo dos defeitos nesses recursos pode não fazer diferença prática de confiabilidade do ponto de vista dos usuários.

3.4 Segurança

Os sistemas de segurança críticos são aqueles em que é essencial que a operação seja sempre segura. Isso significa que o sistema *nunca* causa danos a pessoas ou ao ambiente, mesmo que venha a falhar. Exemplos de sistemas críticos de segurança são sistemas de controle e monitoração de aeronaves, sistemas de controle de processos de indústrias químicas e farmacêuticas e sistemas de controle de automóveis.

O controle de hardware de sistemas críticos de segurança é mais fácil de ser implementado e analisado do que o controle de software. Contudo, atualmente são construídos sistemas de tal complexidade que não podem ser controlados apenas por hardware. Algum controle de software é essencial, devido à necessidade de gerenciar grandes quantidades de sensores e atuadores com regras complexas de controle. Um exemplo dessa complexidade é encontrado em aeronaves militares avançadas instáveis aerodinamicamente. Elas requerem um ajuste contínuo por software de suas superfícies de vôo para garantir que não causem acidentes.

O software crítico de segurança pode ser separado em duas classes:

1. *Software crítico primário de segurança.* O software está embutido como controlador em um sistema. O mau funcionamento desse software pode causar mau funcionamento do hardware, que resulta em danos a pessoas e ao ambiente. Eu enfoco esse tipo de software.
2. *Software crítico secundário de segurança.* Esse software pode resultar em danos indiretamente. Exemplos de tais sistemas são sistemas de auxílio ao projeto de engenharia, cujo mau funcionamento pode resultar em um defeito de projeto no sistema que está sendo desenvolvido. Esse defeito pode causar danos às pessoas se o sistema projetado funcionar mal. Outro exemplo de software crítico secundário de segurança é um banco de dados que armazena detalhes dos medicamentos administrados em pacientes. Erros nesse sistema podem resultar na administração incorreta da dosagem do medicamento.

A confiabilidade e a segurança de sistema estão relacionadas, mas são atributos separados de confiança. Certamente, um sistema crítico de segurança deve ser confiável, deve estar em conformidade com sua especificação e deve operar sem falhas. No entanto, ele pode incorporar recursos de tolerância a defeitos de maneira a fornecer serviços continuamente mesmo que os defeitos ocorram. Contudo, sistemas tolerantes a defeitos não são necessariamente seguros. O software pode ainda apresentar mau funcionamento e causar um comportamento de sistema que resulte em um acidente.

Apesar do fato de nunca podermos ter total certeza de que o sistema está livre de defeitos e é tolerante a eles, existem várias outras razões de por que os sistemas confiáveis de software não são necessariamente seguros:

1. A especificação pode estar incompleta, de forma que não descreve o comportamento exigido do sistema em algumas situações críticas. Uma porcentagem alta de maus funcionamentos do sistema (Nakajo e Kume, 1991; Lutz, 1993) são resultados dos erros de especificação em vez de erros de projeto. Em um estudo de erros em sistemas embutidos, Lutz conclui:

   ... dificuldades com requisitos constituem a causa principal dos erros de software relacionados com segurança, que persistiram até a integração e teste de sistema.

2. Os maus funcionamentos de hardware podem fazer o sistema se comportar de formar imprevisível e expor o software a um ambiente inesperado. Quando os componentes estão próximos a falhar, eles podem se comportar de forma errática e gerar sinais fora das faixas que podem ser tratados pelo software.
3. Os operadores de sistema podem gerar entradas incorretas individualmente, mas que, em certas situações, podem levar a um mau funcionamento do sistema. Um exemplo disso pode ser um mecânico que instrui o software de gerenciamento de dispositivos de uma aeronave para elevar o trem de aterrissagem. O software realizou perfeitamente a instrução do mecânico. Infelizmente, o avião estava no chão — naturalmente, o sistema deveria ter desabilitado o comando, a menos que o avião estivesse no ar.

Um vocabulário especializado foi desenvolvido para discutir os sistemas críticos de segurança e isso é importante para compreender os termos específicos usados. A Tabela 3.2 apresenta algumas definições adaptadas dos termos inicialmente definidos por Leveson (Leveson, 1985).

A chave para garantir a segurança é garantir que o acidente não ocorra ou que as conseqüências sejam mínimas. Isso pode ser conseguido por meio de três maneiras complementares:

Tabela 3.2 Terminologia de segurança.

Termo	Descrição
Acidente (ou desgraça)	Evento ou seqüência de eventos não planejados que resulta em morte ou ferimento de humanos, danos à propriedade ou ao ambiente. Uma máquina controlada por computador que fere seu operador é um exemplo de um acidente.
Perigo	Condição com potencial para causar ou contribuir para um acidente. A falha de um sensor que detecta um obstáculo em frente de uma máquina é um exemplo de perigo.
Dano	Medida de perda resultante de um acidente. Um dano pode variar desde a morte de várias pessoas como resultado de um acidente até ferimentos de pouca importância ou danos à propriedade.
Severidade do perigo	Avaliação do pior dano possível que poderia resultar de determinado perigo. A severidade do perigo pode variar de catastrófica, na qual várias pessoas são mortas, até somente danos de pouca importância.
Probabilidade de perigos	Probabilidade de ocorrência de eventos que criam um risco. Valores de probabilidade tendem a ser arbitrários, mas variam de *provável* (digamos, chance de 1/100 de ocorrência de um risco) a *implausível* (não existem situações concebíveis nas quais o perigo possa ocorrer).
Risco	É a medida da probabilidade de que o sistema causará um acidente. O risco é avaliado considerando-se a probabilidade do perigo, a severidade do perigo e a probabilidade de que o perigo resultará em um acidente.

1. *Prevenção de perigos.* O sistema é projetado de tal modo que os perigos sejam evitados. Por exemplo, um sistema de corte, que requer que o operador pressione dois botões separados simultaneamente para operar a máquina, evita o perigo de as mãos do operador estarem no caminho da lâmina.
2. *Detecção e remoção de perigos.* O sistema é projetado de tal forma que os perigos sejam detectados e removidos antes de causarem um acidente. Por exemplo, um sistema de uma indústria química pode detectar a pressão excessiva e abrir uma válvula de alívio para reduzir a pressão, antes que ocorra uma explosão.
3. *Limitação de danos.* O sistema pode incluir recursos de proteção que minimizem os danos resultantes de um acidente. Por exemplo, um motor de aeronave normalmente inclui um extintor automático de incêndio. Se ocorrer um incêndio, ele poderá ser controlado antes que ameace a aeronave.

Os acidentes geralmente ocorrem quando várias coisas estão erradas ao mesmo tempo. Uma análise de acidentes sérios (Perrow, 1984) sugere que eles foram quase todos devidos à combinação de maus funcionamentos em vez de falhas simples. A combinação não prevista levou às interações que resultaram na falha do sistema. Perrow também sugere que é impossível prever todas as possíveis combinações de maus funcionamentos do sistema e que acidentes constituem uma parte inevitável do uso de sistemas complexos. O software tende a aumentar a complexidade dos sistemas e, portanto, o uso do controle de software *pode* aumentar a probabilidade de acidentes de sistema.

No entanto, o controle e a monitoração de software podem também aprimorar os sistemas de segurança. Os sistemas controlados por software podem monitorar uma faixa mais ampla de condições do que sistemas eletromecânicos. Podem ser adaptados com relativa facilidade. Eles envolvem o uso de hardware de computador, que, além de possuir alta confiabilidade inerente, é fisicamente pequeno e leve. Os sistemas controlados por software podem fornecer intertravamentos sofisticados de segurança. Podem apoiar as estratégias de controle que reduzem a quantidade de tempo que as pessoas precisam passar em ambientes perigosos. Portanto, embora o controle de software possa introduzir mais maneiras de os sistemas não funcionarem bem, ele permite também melhor monitoração e proteção e, por isso, pode aprimorar a segurança do sistema.

Em todos os casos, é importante manter um senso de proporção sobre a segurança do sistema. É impossível criar um sistema totalmente seguro e a sociedade precisa decidir se as conseqüências de um acidente ocasional compensam os benefícios provenientes do uso de tecnologias avançadas. É também uma decisão social e política sobre como liberar os recursos nacionais limitados para reduzir perigos à população como um todo.

3.5 Proteção

A proteção é um atributo que reflete a capacidade do sistema de se proteger de ataques externos acidentais ou propositais. A importância da proteção cresceu, à medida que mais e mais sistemas foram conectados à Internet. As conexões com a Internet fornecem funcionalidades adicionais de sistemas (por exemplo, os clientes podem acessar suas contas bancárias diretamente), mas também significam que o sistema pode ser atacado por pessoas com más intenções. A conexão

com a Internet significa ainda que informações sobre vulnerabilidades específicas do sistema podem ser facilmente disseminadas de tal maneira que mais pessoas podem ser capazes de atacar o sistema. Do mesmo modo, no entanto, a conexão pode aumentar a velocidade da distribuição de *patches* para corrigir essas vulnerabilidades.

Exemplos de ataques podem ser vírus, uso não autorizado de serviços de sistema e modificações não autorizadas do sistema ou de seus dados. A proteção é importante em todos os sistemas críticos. Sem um nível razoável de proteção, a disponibilidade, confiabilidade e segurança do sistema podem ser comprometidas se ataques externos causarem algum dano.

A razão para isso é que todos os métodos para garantia de disponibilidade, confiabilidade e segurança contam com o fato de que o sistema operacional é o mesmo de quando o sistema foi originalmente instalado. Se esse sistema instalado foi comprometido de alguma forma (por exemplo, o software foi modificado para incluir um vírus), os argumentos para confiabilidade e segurança originalmente usados podem não valer mais. O sistema de software pode ter se corrompido e pode se comportar de forma não prevista.

Por outro lado, erros no desenvolvimento de um sistema podem levar às brechas de proteção. Se um sistema não responde a entradas não esperadas ou se os limites de vetores não são verificados, os atacantes podem explorar essas fraquezas para obter acesso ao sistema. Os principais incidentes relacionados à proteção, tais como o *worm* original de Internet (Spafford, 1989) e o *worm Code Red* mais de dez anos atrás (Berghel, 2001), levaram vantagem pelo fato de os programas em C não incluírem a verificação dos limites dos vetores. Eles sobrescreviam parte da memória com códigos que permitiam acessos não autorizados ao sistema.

Obviamente, em alguns sistemas críticos, proteção é a dimensão mais importante de confiança do sistema. Sistemas militares, sistemas de comércio eletrônico e sistemas que envolvem processamento e troca de informações confidenciais devem ser projetados de tal maneira que alcancem um alto nível de proteção. Se um sistema de reserva de companhias aéreas, por exemplo, estiver indisponível, isso causará inconveniência e alguns atrasos na emissão de passagens. No entanto, se o sistema estiver desprotegido e puder aceitar reservas falsas, a companhia aérea poderá ter um grande prejuízo.

Há três tipos de danos que podem ser causados por ataques externos:

1. *Recusa de serviço.* O sistema pode ser forçado a um estado em que seus serviços normais tornem-se indisponíveis. Isso, obviamente, afeta a disponibilidade do sistema.
2. *Corrupção de programa ou dados.* Os componentes de software podem ser alterados de maneira não autorizada. Isso pode afetar o comportamento do sistema e, portanto, sua confiabilidade e segurança. Se o dano for grave, a disponibilidade do sistema pode ser afetada.
3. *Vazamento de informações confidenciais.* As informações gerenciadas pelo sistema podem ser confidenciais e o ataque externo pode expô-las para pessoas não autorizadas. Dependendo do tipo de dados, isso pode afetar a segurança do sistema e permitir ataques posteriores que afetem a disponibilidade ou a confiabilidade.

Como nos outros aspectos de confiança, existe uma terminologia especializada associada à proteção. Alguns termos importantes, conforme discutidos por Pfleeger (Pfleeger, 1997), são definidos na Tabela 3.3.

Existe uma clara analogia, no caso, com alguns termos da terminologia de segurança, tal que uma exposição é análoga a um acidente e uma vulnerabilidade é análoga a um perigo.

Portanto, existem abordagens comparáveis que podem ser usadas para garantir a segurança de acesso a um sistema:

1. *Prevenção de vulnerabilidade.* O sistema é projetado de maneira que as vulnerabilidades não ocorram. Por exemplo, se um sistema não estiver conectado a uma rede pública externa, não existe a possibilidade de um ataque pelos membros do público.

Tabela 3.3 Terminologia de proteção.

Termo	Descrição
Exposição	Possível perda ou dano no sistema computacional. Pode ser perda ou danos nos dados ou pode ser perda de tempo ou esforço, se a recuperação é necessária após uma brecha na proteção.
Vulnerabilidade	Uma fraqueza no sistema baseado em computador que pode ser explorada para causar perda ou dano.
Ataque	Uma exploração da vulnerabilidade do sistema. Geralmente parte de fora do sistema e é uma tentativa deliberada para causar algum dano.
Ameaças	Circunstâncias que têm potencial para causar perda ou dano. Você pode pensar nelas como uma vulnerabilidade do sistema que está sujeita a um ataque.
Controle	Uma medida de proteção que reduz uma vulnerabilidade do sistema. Criptografia pode ser um exemplo de controle que reduz a vulnerabilidade de um sistema fraco de controle de acesso.

2. *Detecção e neutralização de ataque.* O sistema é projetado para detectar vulnerabilidades e removê-las antes que causem uma exposição. Um exemplo de detecção e remoção de vulnerabilidade é o uso de um verificador de vírus que analisa os arquivos recebidos e modifica esses arquivos para remover o vírus.
3. *Limitação de exposição.* As conseqüências de um ataque bem-sucedido são minimizadas. Exemplos da limitação de exposição são back-ups regulares do sistema e uma política de gerenciamento de configuração que permite a recriação de software danificado.

A maioria das vulnerabilidades em sistemas baseados em computadores resulta de falhas humanas mais do que técnicas. As pessoas escolhem senhas fáceis de adivinhar ou escrevem em locais onde podem ser encontradas. Os administradores do sistema cometem erros quando estabelecem controle de acesso ou arquivos de configuração. Para aprimorar a proteção, portanto, precisamos pensar sobre como os sistemas são realmente usados, e não apenas sobre suas características técnicas. Esse assunto será explicado com mais detalhes no Capítulo 30, que aborda a engenharia de proteção, na nova seção sobre Tecnologias Emergentes.

PONTOS-CHAVE

- Em um sistema crítico, a falha pode levar a perdas econômicas significativas, danos físicos ou ameaças à vida humana. As três classes importantes de sistemas críticos são sistemas críticos de segurança, de missão e de negócios.
- A confiança de um sistema de computador é uma propriedade do sistema que reflete o nível de confiança do usuário em relação ao sistema. As dimensões mais importantes da confiança são disponibilidade, confiabilidade, segurança e proteção.
- A disponibilidade de um sistema é a probabilidade de que o sistema será capaz de fornecer serviços aos usuários quando solicitado. Confiabilidade é a probabilidade de que os serviços do sistema serão fornecidos conforme especificados.
- Confiabilidade e disponibilidade são geralmente consideradas como as dimensões mais importantes da confiança. Se um sistema não é confiável, é difícil garantir a segurança e a proteção do sistema, pois elas podem estar comprometidas por falhas de sistema.
- A confiabilidade está relacionada com a probabilidade de um erro que ocorre durante o uso operacional. Um programa pode conter defeitos conhecidos, mas pode ainda ser considerado confiável por seus usuários. Eles podem nunca usar os recursos do sistema afetados por esses defeitos.
- A segurança de um sistema é um atributo que reflete a habilidade de o sistema operar, normal ou anormalmente, sem ameaçar as pessoas ou o ambiente.
- A proteção é importante para todos os sistemas críticos. Sem um nível razoável de proteção, a disponibilidade, confiabilidade e segurança do sistema podem ser comprometidas se ataques externos causarem algum dano ao sistema.
- Para aprimorar a confiança, você precisa considerar uma abordagem sociotécnica para o projeto do sistema, levando em conta as pessoas no sistema, assim como o hardware e o software.

LEITURAS SUGERIDAS

"The evolution of information assurance". Excelente artigo que explica a necessidade de proteger informações críticas contra acidentes e ataques em uma organização. (R. Cummings, *IEEE computer*, 35 (12), dezembro de 2002.)

Practical design of safety-critical computer systems. Uma visão geral de sistemas críticos de segurança que explica assuntos de segurança e que considera os sistemas não meramente sob uma perspectiva de software. (W. R. Dunn, 2002, Reliability Press.)

Secrets and lies: digital security in a networked world. Um livro excelente e de muito fácil leitura sobre proteção de computadores que aborda o assunto sob uma perspectiva sociotécnica. (B. Schneier, 2000, John Wiley & Sons.)

"Survivability: protecting your critical systems". Uma introdução acessível para o tópico de sobrevivência e por que é importante. (R. Ellison et al., *IEEE internet computing*, novembro/dezembro de 1999.)

Computer-related risks. Uma coleção tirada de um fórum de incidentes sobre riscos de Internet ocorridos em sistemas automatizados. Ela mostra o que pode realmente dar errado em sistemas relacionados com a segurança. (P. G. Neumann, 1995, Addison-Wesley.)

EXERCÍCIOS

3.1 Quais são os três principais tipos de sistemas críticos? Explique as diferenças entre eles.

3.2 Sugira seis razões por que a confiança é importante para sistemas críticos.

3.3 Quais são as dimensões mais importantes de confiança no sistema?

3.4 Por que o custo para garantir a confiança é exponencial?

3.5 Justificando sua resposta, sugira quais atributos de confiança são provavelmente os mais críticos para os seguintes sistemas:

- Um servidor de Internet provido por um ISP com milhares de clientes
- Um bisturi controlado por computador usado em cirurgias
- Um sistema de controle direcional usado no veículo lançador de satélites
- Um sistema de gerenciamento de finanças pessoais baseado em Internet

3.6 Identifique seis produtos destinados ao consumidor que contêm ou que podem conter sistemas críticos de software no futuro.

3.7 Confiabilidade e segurança são atributos de confiança relacionados, mas distintos. Descreva a distinção mais importante entre os atributos e explique por que é possível um sistema confiável ser não seguro e vice-versa.

3.8 Em um sistema médico projetado para emitir radiação para tratamento de tumores, sugira um perigo que pode vir a ocorrer e proponha um recurso de software que possa garantir que o perigo identificado não resulte em um acidente.

3.9 Explique por que existe uma relação forte entre disponibilidade e proteção de sistema.

3.10 Em termos de proteção de computador, explique as diferenças entre um ataque e uma ameaça.

3.11 É ético para um engenheiro concordar em liberar um sistema de software com defeitos conhecidos para um cliente? Isso faz alguma diferença se o cliente for avisado sobre a existência desses defeitos antecipadamente? Seria razoável fazer reclamações sobre a confiabilidade do software em tais circunstâncias?

3.12 Como especialista em proteção de computador, você foi contatado por uma organização que faz campanha pelos direitos de vítimas de tortura e lhe foi solicitado que ajude a organização a conseguir um acesso não autorizado aos sistemas de computador de uma empresa norte-americana. Isso irá auxiliá-los a confirmar ou negar que essa empresa está vendendo equipamento usado diretamente na tortura de presos políticos. Explique os dilemas éticos que esse pedido traz e como você reagiria a ele.

4

Processos de software

Objetivos

O objetivo deste capítulo é apresentar a idéia de um processo de software — um conjunto coerente de atividade para produção de software. Após ler este capítulo, você compreenderá:

- o conceito de processos de software e de modelos de processo de software;
- três modelos de processo de software e quando eles podem ser utilizados;
- em termos gerais, as atividades realizadas em engenharia de requisitos de software, o desenvolvimento de software, a realização de testes e a evolução de software;
- como o Rational Unified Process integra a boa prática de processo de software para criar um modelo de processo genérico e moderno;
- a tecnologia CASE, usada para apoiar as atividades de processo de software.

Conteúdo

4.1 Modelos de processo de software
4.2 Iteração de processo
4.3 Atividades de processo
4.4 O Rational Unified Process
4.5 Engenharia de Software Auxiliada por Computador

Um processo de software é um conjunto de atividades que leva à produção de um produto de software. Essas atividades podem envolver o desenvolvimento de software propriamente dito, usando uma linguagem de programação como Java ou C. Cada vez mais, no entanto, novo software é desenvolvido com a ampliação e a modificação de sistemas existentes e de configuração e integração de software comercial ou componentes de sistema.

Os processos de software são complexos e, como todos os processos intelectuais e criativos, dependem de julgamento humano. Por causa da necessidade de utilizar o julgamento e a criatividade, as tentativas de automatização dos processos de software têm tido sucesso limitado. As ferramentas de engenharia de software auxiliada por computador (CASE — Computer-Aided Software Engineering) (explicadas na Seção 4.5) podem apoiar algumas atividades de processo. No entanto, não existe possibilidade, pelo menos nos próximos anos, de uma automação mais extensa, na qual o software assuma o projeto criativo, liberando os engenheiros envolvidos no processo de software.

Uma razão para a limitada eficiência das ferramentas CASE decorre da imensa diversidade dos processos de software. Não existe um processo ideal, e várias organizações desenvolveram abordagens inteiramente diferentes para o desenvolvimento de software. Os processos evoluíram para explorar as capacidades das pessoas em uma organização e as características específicas dos sistemas que estão sendo desenvolvidos. No caso de alguns sistemas, como os sistemas críticos, é necessário um processo de desenvolvimento muito estruturado. Nos sistemas de negócios, com requisitos que mudam rapidamente, um processo flexível e ágil é provavelmente mais eficaz.

Embora existam muitos processos de software diferentes, algumas atividades fundamentais são comuns a todos eles, como:

1. *Especificação de software.* A funcionalidade do software e as restrições sobre sua operação devem ser definidas.
2. *Projeto e implementação de software.* O software que atenda à especificação deve ser produzido.
3. *Validação de software.* O software deve ser validado para garantir que ele faça o que o cliente deseja.
4. *Evolução de software.* O software deve evoluir para atender às necessidades mutáveis do cliente.

Explico essas atividades brevemente neste capítulo e com mais detalhes nas partes posteriores deste livro.

Embora não exista um processo de software 'ideal', existe espaço para aprimoramento do processo de software em várias organizações. Os processos podem incluir técnicas obsoletas ou não tirar vantagem das melhores práticas na engenharia de software industrial. De fato, muitas organizações ainda não se beneficiam dos métodos de engenharia de software em seu desenvolvimento de software.

Os processos de software podem ser aprimorados por meio da padronização de processo, na qual a diversidade de processos de software ao longo da organização é reduzida. Isso promove o aprimoramento da comunicação e redução no tempo de treinamento e faz com que o apoio ao processo automatizado seja mais econômico. A padronização é também um passo inicial importante na introdução de novos métodos e técnicas de engenharia de software e também nas boas práticas de engenharia de software. O aprimoramento do processo de software será abordado no Capítulo 28 com mais detalhes.

4.1 Modelos de processo de software

Conforme expliquei no Capítulo 1, um modelo de processo de software é uma representação abstrata de um processo de software. Cada modelo de processo representa um processo sob determinada perspectiva e, dessa forma, fornece somente informações parciais sobre esse processo. Nesta seção, apresento uma série de modelos de processo muito genéricos (às vezes chamados de *paradigmas de processo*) sob a perspectiva da arquitetura. Isto é, vemos o framework do processo, mas não os detalhes de atividades específicas.

Esses modelos genéricos não são descrições definitivas de processos de software. Ao contrário, são abstrações do processo que podem ser usadas para explicar diferentes abordagens para o desenvolvimento de software. Eles podem ser considerados como frameworks de processo que podem ser ampliadas e adaptadas para criar processos mais específicos de engenharia de software.

Os modelos de processos discutidos neste capítulo são:

1. *O modelo em cascata.* Considera as atividades fundamentais do processo, compreendendo especificação, desenvolvimento, validação e evolução, e as representa como fases de processo separadas, tais como especificação de requisitos, projeto de software, implementação, teste e assim por diante.
2. *Desenvolvimento evolucionário.* Esta abordagem intercala as atividades de especificação, desenvolvimento e validação. Um sistema inicial é desenvolvido rapidamente baseado em especificações abstratas. Este sistema é, então, refinado com as entradas do cliente para produzir um sistema que satisfaça as necessidades do cliente.
3. *Engenharia de software baseada em componentes.* Esta abordagem baseia-se na existência de um número significativo de componentes reusáveis. O processo de desenvolvimento do sistema enfoca a integração desses componentes, em vez de desenvolvê-los a partir do zero.

Esses três modelos genéricos de processo são amplamente usados na prática atual de engenharia de software. Eles não são mutuamente exclusivos e freqüentemente são usados em conjunto, especialmente para desenvolvimentos de sistemas de grande porte. De fato, o Rational Unified Process, que abordo na Seção 4.4, combina os elementos de todos esses modelos. Os subsistemas contidos em um sistema maior podem ser desenvolvidos usando diferentes abordagens. Portanto, embora seja conveniente explicar esses modelos separadamente, você deve entender que, na prática, freqüentemente eles são combinados.

Todos os tipos de variações desses processos genéricos foram propostos e podem ser usados em algumas organizações. A variação mais importante é, provavelmente, o desenvolvimento formal de sistemas, no qual é criado um modelo mate-

mático formal de um sistema. Esse modelo é, então, modificado por meio de transformações matemáticas que preservam sua consistência, em código executável.

O exemplo mais conhecido do processo formal de desenvolvimento é o processo *Cleanroom*, originalmente desenvolvido pela IBM (Mills, et al., 1987; Selby, et al., 1987; Linger, 1994; Prowell, et al., 1999). No processo Cleanroom, cada incremento de software é especificado formalmente e essa especificação é transformada em uma implementação. A correção de software é demonstrada por meio de uma abordagem formal. Não existe teste de defeitos no processo e o teste do sistema concentra-se na avaliação da confiabilidade.

Tanto a abordagem Cleanroom quanto uma outra abordagem baseada no método B (Wordsworth, 1996) são particularmente adequadas para desenvolvimento de sistemas com requisitos rigorosos de segurança, confiabilidade ou proteção. A abordagem formal simplifica a produção de um caso de segurança ou proteção, que demonstra aos clientes ou aos órgãos de certificação que o sistema realmente atende aos requisitos de segurança ou proteção.

Fora desses domínios especializados, os processos baseados em transformações formais não são largamente usados. Eles exigem um conhecimento especializado e, na realidade, para a maioria dos sistemas, esse processo não oferece vantagens significativas de custo ou de qualidade sobre outras abordagens de desenvolvimento do sistema.

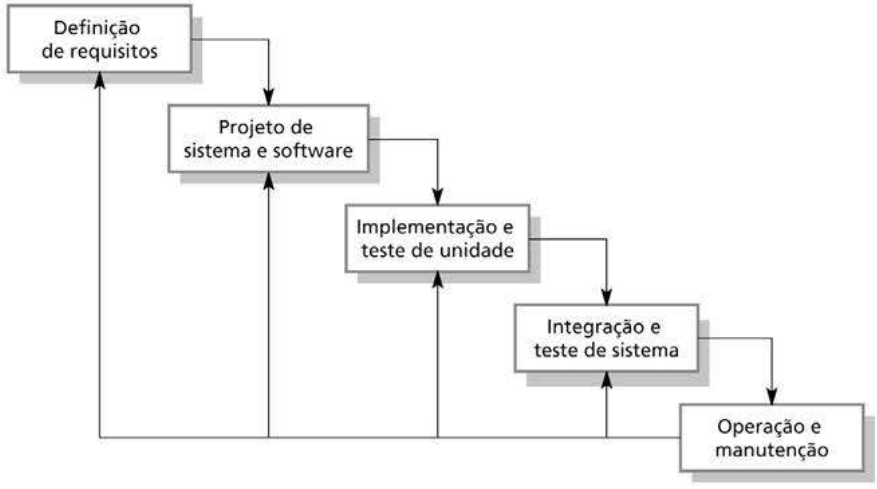
4.1.1 O modelo em cascata

O primeiro modelo de processo de desenvolvimento de software publicado originou-se de processos mais gerais de engenharia de sistema (Royce, 1970). Isso é mostrado na Figura 4.1. Devido ao encadeamento de uma fase com outra, esse modelo é conhecido como modelo em cascata ou ciclo de vida do software. Os principais estágios do modelo demonstram as atividades fundamentais de desenvolvimento:

1. *Análise e definição de requisitos.* Os serviços, restrições e objetivos do sistema são definidos por meio de consulta aos usuários do sistema. Eles são, portanto, definidos detalhadamente e servem como uma especificação de sistema.
2. *Projeto de sistema e software.* O processo de projeto de sistema divide os requisitos em sistemas de hardware ou de software. Ele estabelece uma arquitetura geral do sistema. O projeto de software envolve a identificação e a descrição das abstrações fundamentais do sistema de software e suas relações.
3. *Implementação e teste de unidade.* Durante esse estágio, o projeto de software é realizado como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atende à sua especificação.
4. *Integração e teste de sistema.* As unidades individuais de programa ou os programas são integrados e testados como um sistema completo para garantir que os requisitos de software foram atendidos. Após os testes, o sistema de software é liberado para o cliente.
5. *Operação e manutenção.* Geralmente (embora não necessariamente) esta é a fase mais longa do ciclo de vida. O sistema é instalado e colocado em operação. A manutenção envolve a correção de erros não detectados nos estágios anteriores do ciclo de vida, no aprimoramento da implementação das unidades de sistema e na ampliação dos serviços de sistema à medida que novos requisitos são identificados.

Em princípio, o resultado de cada fase consiste de um ou mais documentos aprovados ('assinados'). A fase seguinte não deve começar antes que a fase anterior tenha terminado. Na prática, esses estágios se sobrepõem e trocam informações

Figura 4.1
Ciclo de vida de software.



entre si. Durante o projeto, são identificados problemas com requisitos; durante a codificação, são encontrados problemas de projeto e assim por diante. O processo de software não é um modelo linear simples; envolve uma seqüência de iterações das atividades de desenvolvimento.

Devido aos custos de produção e aprovação de documentos, as iterações são onerosas e envolvem um 'retrabalho' significativo. Portanto, após um pequeno número de iterações, é normal suspender partes do desenvolvimento, como a especificação, e prosseguir com os estágios posteriores do desenvolvimento. Os problemas são resolvidos posteriormente, ignorados ou reprogramados. O congelamento prematuro de requisitos pode significar que o sistema não fará o que o usuário deseja. Isso pode também levar a sistemas mal estruturados, pois os problemas de projeto foram contornados por meio de artifícios de implementação.

Durante a fase final do ciclo de vida (operação e manutenção), o software é colocado em uso. Erros e omissões nos requisitos originais de software são descobertos. Os erros de programação e projeto emergem e a necessidade de novas funcionalidades é identificada. O sistema deve, portanto, evoluir para permanecer útil. Essas mudanças (manutenção de software) podem implicar repetição de estágios anteriores do processo.

As vantagens do modelo em cascata consistem na documentação produzida em cada fase e sua aderência a outros modelos de processo de engenharia. Seu maior problema é a divisão inflexível do projeto em estágios distintos. Os compromissos devem ser assumidos no estágio inicial do processo, o que torna difícil reagir às mudanças de requisitos do usuário.

Portanto, o modelo em cascata deve ser usado apenas quando os requisitos forem bem compreendidos e houver pouca probabilidade de mudanças radicais durante o desenvolvimento do sistema. No entanto, o modelo em cascata reflete o tipo de modelo de processo usado em outros projetos de engenharia. Conseqüentemente, processos de software baseados nessa abordagem são ainda usados para desenvolvimento de software, particularmente quando fazem parte de um projeto maior de engenharia do sistema.

4.1.2 Desenvolvimento evolucionário

O desenvolvimento evolucionário baseia-se na idéia de desenvolvimento de uma implementação inicial, expondo o resultado aos comentários do usuário e refinando esse resultado por meio de várias versões até que seja desenvolvido um sistema adequado (Figura 4.2). As atividades de especificação, desenvolvimento e validação são intercaladas, em vez de serem separadas, com feedback rápido que permeia as atividades.

Existem dois tipos fundamentais de desenvolvimento evolucionário:

1. *Desenvolvimento exploratório,* no qual o objetivo do processo é trabalhar com o cliente para explorar os requisitos e entregar um sistema final. O desenvolvimento começa com as partes do sistema compreendidas. O sistema evolui por meio da adição de novas características propostas pelo cliente.
2. *Prototipação throwaway,* na qual o objetivo do processo de desenvolvimento evolucionário é compreender os requisitos do cliente e, a partir disso, desenvolver melhor definição de requisitos para o sistema. O protótipo se concentra na experimentação dos requisitos mal compreendidos do cliente.

Uma abordagem evolucionária para desenvolvimento de software é freqüentemente mais eficaz do que a abordagem em cascata na produção de sistemas que atendam às necessidades imediatas dos clientes. A vantagem de um processo de software baseado na abordagem evolucionária é que a especificação pode ser desenvolvida de forma incremental. À medida que os usuários compreendem melhor seu problema, isso pode ser refletido no sistema de software. No entanto, do ponto de vista da engenharia e do gerenciamento, a abordagem evolucionária tem dois problemas:

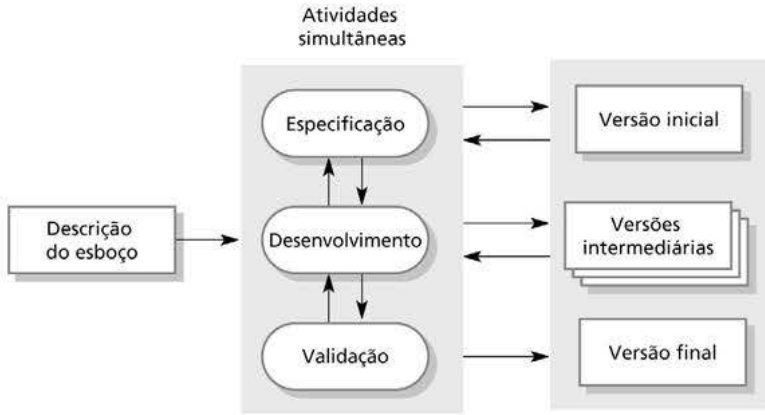
1. *O processo não é visível.* Os gerentes precisam de produtos regulares para medir o progresso. Se os sistemas são desenvolvidos rapidamente, não é viável economicamente produzir documentos que reflitam cada versão do sistema.
2. *Os sistemas são freqüentemente mal estruturados.* A mudança contínua tende a corromper a estrutura do software. A incorporação de mudanças de software torna-se cada vez mais difícil e onerosa.

Para sistemas de pequeno e médio porte (até 500 mil linhas de código), penso que a abordagem evolucionária seja o melhor método de desenvolvimento. Os problemas de desenvolvimento evolucionário tornam-se particularmente graves para sistemas complexos de grande porte e de longo ciclo de vida, nos quais diversas equipes desenvolvem diferentes partes do sistema. É difícil estabelecer uma arquitetura estável do sistema usando essa abordagem, o que torna difícil integrar as contribuições das equipes.

Para sistemas de grande porte, recomendo um processo misto que incorpore as melhores características dos modelos de desenvolvimento em cascata e evolucionário. Isso pode envolver o desenvolvimento de um protótipo throwaway, usando uma abordagem evolucionária para resolver as incertezas na especificação do sistema. Você pode, portanto, reimplementar o sistema usando uma abordagem mais estruturada. As partes do sistema bem compreendidas podem ser especificadas e desenvolvidas

Figura 4.2

Desenvolvimento evolucionário.



usando um processo baseado em cascata. Outras partes do sistema, tais como interface com o usuário, que são difíceis de serem especificadas antecipadamente, devem ser sempre desenvolvidas usando uma abordagem de programação exploratória.

Os processos de desenvolvimento evolucionário e o apoio ao processo são abordados mais detalhadamente no Capítulo 17, junto com a prototipação de sistema e o desenvolvimento ágil de software. O desenvolvimento evolucionário está também incorporado ao Rational Unified Process explicado mais adiante neste capítulo.

4.1.3 Engenharia de software baseada em componentes

Na maioria dos projetos de software, existe algum reuso de software. Isso ocorre geralmente de maneira informal, quando as pessoas que trabalham no projeto conhecem os projetos ou os códigos similares aos necessários. Elas procuram por esses produtos, os modificam e os incorporam ao sistema. Na abordagem evolucionária, descrita na Seção 4.1.2, o reuso é freqüentemente essencial para o desenvolvimento rápido do sistema.

Esse reuso informal ocorre independentemente do processo de desenvolvimento usado. No entanto, nos últimos anos, uma abordagem para desenvolvimento de software, denominada engenharia de software, baseada em componentes (CBSE — *Component-Based Software Engineering*) e que conta com reuso, tem emergido e se tornado cada vez mais utilizada. Apresento brevemente essa abordagem nesta seção e mais detalhadamente no Capítulo 19.

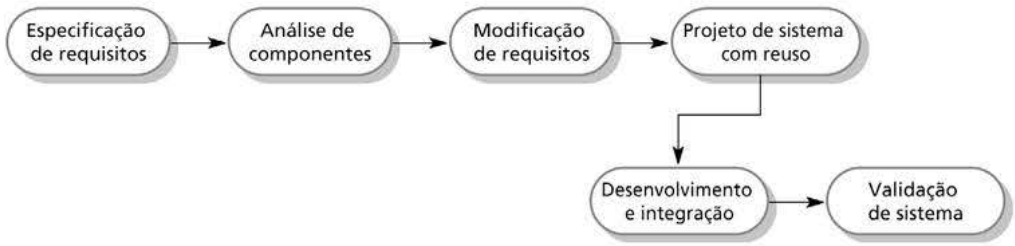
A abordagem orientada a reuso depende de uma grande base de componentes de software reusáveis e algum framework de integração desses componentes. Algumas vezes, esses componentes são sistemas comerciais independentes (COTS ou Commercial Off-The-Shelf Systems) que podem fornecer funcionalidade específica, como a formatação de texto ou um cálculo numérico. O modelo genérico de processo para CBSE é mostrado na Figura 4.3.

Embora o estágio de especificação inicial de requisitos e o estágio de validação sejam comparáveis a outros processos, os estágios intermediários em um processo orientado a reuso são diferentes. Esses estágios são:

1. *Análise de componentes.* Dada uma especificação de requisitos, é feita uma busca pelos componentes para implementar essa especificação. Geralmente, não existe uma correspondência exata e os componentes que podem ser usados fornecem apenas parte da funcionalidade necessária.
2. *Modificação de requisitos.* Durante esse estágio, os requisitos são analisados usando as informações sobre os componentes encontrados. Eles são modificados para refletir os componentes disponíveis. Quando as modificações são impossíveis, a atividade de análise de componentes pode ser novamente realizada para procurar soluções alternativas.
3. *Projeto de sistema com reuso.* Durante este estágio, o framework do sistema é projetado ou um framework existente é reusada. Os projetistas levam em consideração os componentes reusados, organizando o framework para eles. Pode ser necessário projetar algum software novo caso os componentes reusáveis não estejam disponíveis.

Figura 4.3

Engenharia de software baseado em componentes.



4. *Desenvolvimento e integração.* O software que não pode ser adquirido externamente é desenvolvido e os componentes e os sistemas COTS são integrados para criar o novo sistema. A integração de sistema, neste modelo, pode ser parte do processo de desenvolvimento, em vez de ser uma atividade separada.

A engenharia de software baseada em componentes tem a vantagem óbvia de reduzir a quantidade de software a ser desenvolvido e, dessa maneira, reduzir os custos e riscos. Isso também leva geralmente a uma entrega mais rápida de software. No entanto, compromissos com os requisitos são inevitáveis e isso pode levar a um sistema que não atenda às reais necessidades dos usuários. Além disso, algum controle sobre a evolução do sistema será perdido se novas versões dos componentes reusáveis não estiverem sob controle da organização que as utiliza.

A CBSE tem muito em comum com uma abordagem emergente para desenvolvimento de software baseada na integração de Web services de vários fornecedores. Essa abordagem de desenvolvimento centrada em serviços será explicada no Capítulo 12.

4.2 Iteração de processo

A mudança é inevitável em todos os projetos de grande porte. Os requisitos de sistema mudam à medida que a empresa que está adquirindo o sistema responde às pressões externas. As prioridades de gerenciamento mudam. Quando novas tecnologias tornam-se disponíveis, projetos e implementações mudam. Isso significa que o processo de software não é de execução única; pelo contrário, as atividades de processo são repetidas regularmente à medida que o sistema é retrabalhado, em resposta às solicitações de mudança.

O desenvolvimento iterativo é tão fundamental ao software que dedico um capítulo completo ao assunto mais adiante no livro (Capítulo 17). Nesta seção, apresento o assunto por meio da descrição de dois modelos de processo projetados explicitamente para apoiar a iteração de processo:

1. *Entrega incremental.* A especificação, o projeto e a implementação de software são divididos em uma série de incrementos desenvolvidos um de cada vez.
2. *Desenvolvimento espiral.* O desenvolvimento do sistema evolui em espiral para fora a partir de um esboço inicial até o sistema final.

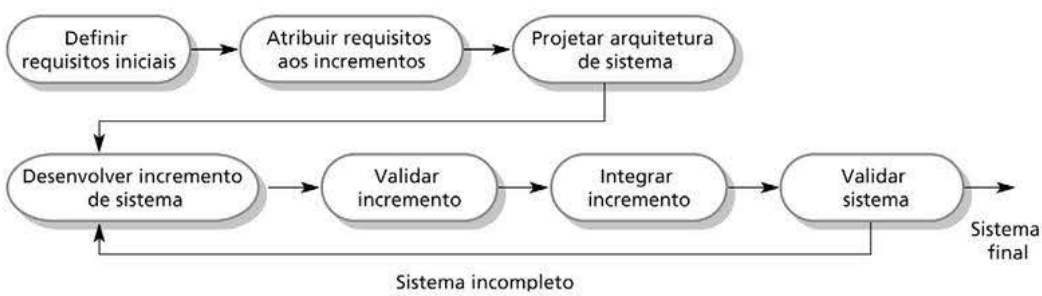
A essência dos processos iterativos é que a especificação é desenvolvida conjuntamente com o software. No entanto, isso conflita com o modelo de aquisição de várias organizações, em que a especificação completa é parte do contrato de desenvolvimento do sistema. Na abordagem incremental, não existe uma especificação completa de software, até que o incremento final seja especificado. Isso requer um novo formato de contrato, aos quais grandes clientes, tais como os órgãos governamentais, podem encontrar dificuldades para se adequar.

4.2.1 Entrega incremental

O modelo em cascata de desenvolvimento requer que os clientes de um sistema se comprometam com um conjunto de requisitos antes do início do projeto e o projetista se comprometa com estratégias específicas de projeto antes da implementação. Mudanças nos requisitos requerem retrabalho dos requisitos, do projeto e da implementação. Contudo, a separação e implementação do projeto deve levar a sistemas bem documentados sujeitos a mudanças. Por outro lado, uma abordagem evolucionária para o desenvolvimento permite que as decisões de requisitos e projeto sejam postergadas, mas também podem levar a um software mal estruturado e difícil de compreender e manter.

A entrega incremental (Figura 4.4) é uma abordagem intermediária que combina as vantagens desses modelos. Em um processo de desenvolvimento incremental, o cliente identifica, em linhas gerais, os serviços a serem fornecidos pelo sistema.

Figura 4.4
Entrega incremental.



Eles identificam quais serviços são mais importantes e quais são menos importantes. Assim, um número de incrementos de entrega é definido, com cada incremento fornecendo um subconjunto das funcionalidades do sistema. A alocação de serviços aos incrementos depende da prioridade do serviço, com os serviços de prioridade mais alta sendo entregues primeiro.

Após a identificação dos incrementos do sistema, os requisitos dos serviços a serem entregues no primeiro incremento são definidos detalhadamente e ele é desenvolvido. Durante o desenvolvimento, pode ser realizada a análise dos próximos requisitos para os incrementos posteriores, mas não são aceitas mudanças de requisitos para o incremento atual.

Depois que um incremento for concluído e entregue, os clientes poderão colocá-lo em operação. Isso significa que eles têm, com antecedência, a entrega de parte da funcionalidade do sistema. Eles podem experimentar o sistema e isso ajuda a conhecer os requisitos dos incrementos posteriores e das versões posteriores do atual. À medida que novos incrementos são concluídos, eles são integrados aos já existentes, de tal forma que a funcionalidade do sistema é aprimorada a cada incremento entregue. Os serviços comuns podem ser implementados no início do processo ou podem ser implementados de forma incremental, conforme a funcionalidade for exigida por um incremento.

O processo de desenvolvimento incremental tem uma série de vantagens:

1. Os clientes não precisam esperar até a entrega do sistema inteiro para se beneficiarem dele. O primeiro incremento satisfaz os requisitos mais críticos e, dessa forma, é possível usar o software imediatamente.
2. Os clientes podem usar os incrementos iniciais como protótipos e ganhar experiência, obtendo informações sobre os requisitos dos incrementos posteriores do sistema.
3. Existe um risco menor de falha geral do projeto. Embora possam ser encontrados problemas em alguns incrementos, é provável que alguns sejam entregues com sucesso aos clientes.
4. Como os serviços de prioridade mais alta são entregues primeiro, e os incrementos posteriores são integrados a eles, é inevitável que os serviços mais importantes de sistema recebam mais testes. Isso significa que os clientes têm menor probabilidade de encontrar falhas de software nas partes mais importantes do sistema.

No entanto, existem problemas com a entrega incremental. Os incrementos devem ser relativamente pequenos (não mais que 20 mil linhas de código) e cada um deve entregar alguma funcionalidade de sistema. Pode ser difícil mapear os requisitos do cliente em incrementos de tamanho adequado. Além disso, a maior parte dos sistemas requer um conjunto de recursos básicos usados por diferentes partes do sistema. Como os requisitos não são definidos detalhadamente até que um incremento seja implementado, pode ser difícil identificar os recursos comuns exigidos por todos os incrementos.

Uma variante dessa abordagem incremental, denominada *extreme programming,* foi desenvolvida (Beck, 2000). Ela se baseia no desenvolvimento e na entrega de incrementos muito pequenos de funcionalidade, envolvimento do cliente no processo, aprimoramento constante de código e programação em pares. Eu explico a *extreme programming* e os outros métodos, denominados de métodos ágeis, no Capítulo 17.

4.2.2 Desenvolvimento em espiral

O modelo em espiral do processo de software (Figura 4.5) foi originalmente proposto por Boehm (Boehm, 1988). Em vez de representar o processo de software como uma seqüência de atividades com algum retorno entre uma atividade e outra, o processo é representado como uma espiral. Cada loop na espiral representa uma fase do processo de software. Dessa forma, o loop mais interno pode estar relacionado à viabilidade do sistema; o próximo loop, à definição de requisitos; o próximo, ao projeto de sistema e assim por diante.

Cada loop na espiral está dividido em quatro setores:

1. *Definição de objetivos.* Os objetivos específicos dessa fase do projeto são definidos. As restrições sobre o processo e o produto são identificadas e um plano detalhado de gerenciamento é elaborado. Os riscos de projeto são identificados. Dependendo disso, estratégias alternativas podem ser planejadas.
2. *Avaliação e redução de riscos.* Para cada risco de projeto identificado, uma análise detalhada é realizada. Providências são tomadas para reduzir o risco. Por exemplo, se houver risco de que os requisitos não sejam apropriados, um protótipo do sistema poderá ser desenvolvido.
3. *Desenvolvimento e validação.* Após a avaliação de risco, um modelo de desenvolvimento para o sistema é selecionado. Por exemplo, se os riscos da interface com o usuário forem dominantes, um modelo de desenvolvimento apropriado pode ser a prototipação evolucionária. Se os riscos de segurança constituírem a principal consideração, o desenvolvimento baseado em transformações formais pode ser o mais apropriado e assim por diante. O modelo em cascata pode ser o modelo de desenvolvimento mais apropriado se o principal risco identificado for a integração de subsistemas.
4. *Planejamento.* O projeto é revisado e uma decisão é tomada para prosseguimento ao próximo loop da espiral. Se a decisão for pelo prosseguimento, serão elaborados planos para a próxima fase do projeto.

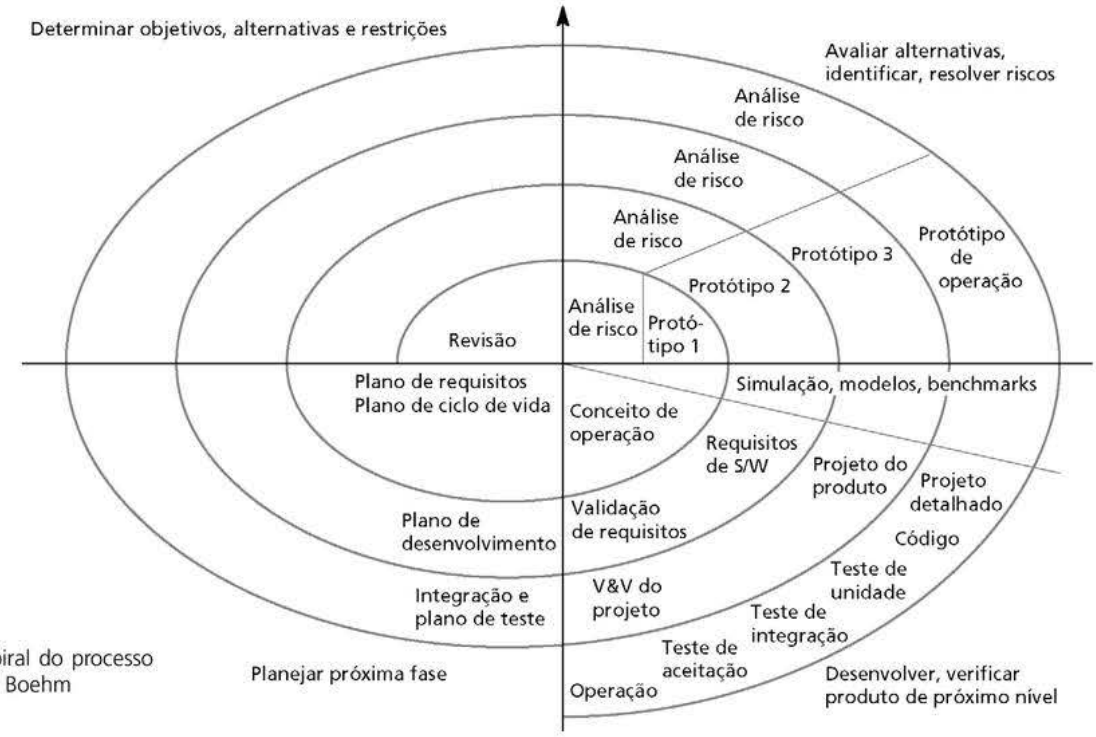


Figura 4.5

Modelo em espiral do processo de software de Boehm (©*IEEE*, 1988).

A principal diferença entre o modelo em espiral e os outros modelos do processo de software é o reconhecimento explícito do risco no modelo em espiral. Informalmente, risco significa simplesmente algo que pode dar errado. Por exemplo, se a intenção for usar uma nova linguagem de programação, um risco é que os compiladores disponíveis não sejam confiáveis ou não produzam código-objeto suficientemente eficaz. Os riscos podem causar problemas no projeto, tal como ultrapassar o cronograma e os custos; por isso, a minimização dos riscos é uma atividade de gerenciamento de projeto muito importante. O gerenciamento de riscos, uma parte essencial de gerenciamento de projetos, é abordada no Capítulo 5.

Um ciclo da espiral começa com a elaboração de objetivos, como desempenho e funcionalidade. Os caminhos alternativos para alcançar esses objetivos e as restrições impostas sobre cada um deles são, então, enumerados. Cada alternativa é avaliada em relação a cada objetivo e as fontes de riscos de projeto são identificadas. O próximo passo é resolver esses riscos por meio de atividades de coleta de informações, tais como análise mais detalhada, prototipação e simulação. Após a avaliação dos riscos, é realizada uma parte do desenvolvimento, seguida pela atividade de planejamento para a próxima fase do processo.

4.3 Atividades de processo

As quatro atividades básicas do processo — especificação, desenvolvimento, validação e evolução — são organizadas de modo diferente nos diversos processos de desenvolvimento. No modelo em cascata, as atividades são organizadas em seqüência, ao passo que, no desenvolvimento evolucionário, elas são intercaladas. Como essas atividades são realizadas depende do tipo de software, pessoas e estruturas organizacionais envolvidas. Não existe uma forma certa ou errada de organizar essas atividades e o meu objetivo nesta seção é simplesmente apresentar a você como elas podem ser organizadas.

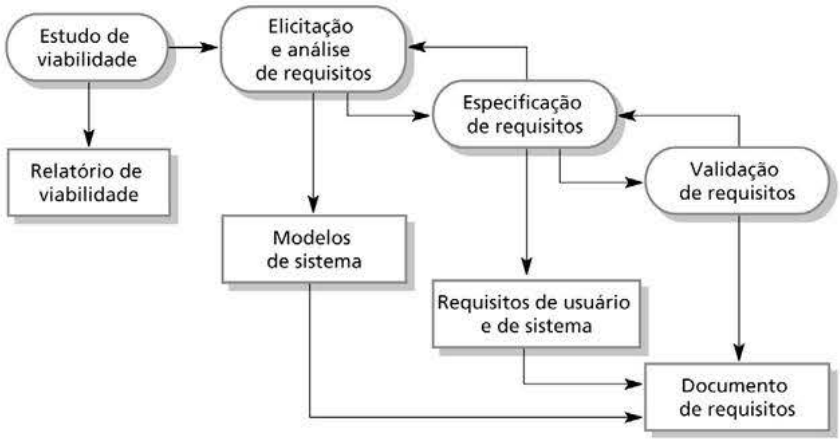
4.3.1 Especificação de software

A especificação de software ou engenharia de requisitos é o processo para compreender e definir quais serviços são necessários e identificar as restrições de operação e de desenvolvimento do sistema. A engenharia de requisitos é um estágio particularmente crítico do processo de software, pois os erros nesse estágio conduzem inevitavelmente a problemas posteriores no projeto e na implementação do sistema.

O processo da engenharia de requisitos é mostrado na Figura 4.6. Esse processo leva à produção de um documento de requisitos, que é a especificação do sistema. Os requisitos são geralmente apresentados em dois níveis de detalhes neste documento. Os usuários finais e os clientes precisam de uma declaração de requisitos de alto nível; os projetistas de sistema precisam de uma especificação de sistema mais detalhada.

Figura 4.6

Processo de engenharia de requisitos.



Existem quatro fases principais no processo de engenharia de requisitos:

1. *Estudo de viabilidade.* É feita uma avaliação para verificar se as necessidades dos usuários identificadas podem ser satisfeitas por meio das tecnologias atuais de software e hardware. O estudo considera se o sistema proposto terá custo adequado do ponto de vista comercial e se pode ser desenvolvido dentro das restrições de orçamento existente. O estudo de viabilidade deve ser relativamente barato e rápido. O resultado deve fornecer informações para a tomada de decisão quanto a prosseguir para uma análise mais detalhada.
2. *Elicitação e análise de requisitos.* É o processo de derivação de requisitos de sistema através da observação de sistemas existentes, discussões com usuários potenciais e compradores, análise de tarefas etc. Isso pode envolver o desenvolvimento de um ou mais modelos de sistema e protótipos. Eles ajudam o analista a compreender o sistema a ser especificado.
3. *Especificação de requisitos.* Atividade de traduzir as informações coletadas durante a atividade de análise em um documento que define um conjunto de requisitos. Devem ser incluídos dois tipos de requisitos nesse documento. *Requisitos de usuário* são declarações abstratas dos requisitos de sistema para o cliente e os seus usuários finais; *requisitos de sistema* constituem uma descrição mais detalhada da funcionalidade a ser fornecida.
4. *Validação de requisitos.* Essa atividade verifica os requisitos em relação ao realismo, consistência e abrangência. Durante esse processo, erros no documento de requisitos são inevitavelmente descobertos. Devem, então, ser feitas modificações para corrigir esses problemas.

Naturalmente, as atividades do processo de requisitos não são realizadas simplesmente em uma seqüência estrita. A análise de requisitos continua durante a definição e a especificação, e novos requisitos aparecem ao longo do processo. Portanto, as atividades de análise, definição e especificação são intercaladas. Em métodos ágeis, como, por exemplo, a *extreme programming*, os requisitos são desenvolvidos de forma incremental, de acordo com as prioridades do usuário, e a elicitação de requisitos provém de usuários que fazem parte da equipe de desenvolvimento.

4.3.2 Projeto e implementação de software

O estágio de implementação do desenvolvimento de software é o processo de conversão de uma especificação de sistema em um sistema executável. Ele sempre envolve os processos de projeto e de programação de software, mas, se uma abordagem evolucionária for usada, pode também envolver o refinamento da especificação de software.

Um projeto de software é a descrição da estrutura de software a ser implementada, dos dados que são partes do sistema, das interfaces entre os componentes do sistema e, às vezes, dos algoritmos usados. Os projetistas não chegam ao projeto final imediatamente, mas desenvolvem o projeto iterativamente por meio de várias versões. O processo de projeto envolve a inclusão de formalidades e detalhes, uma vez que o projeto é desenvolvido por meio de revisões constantes dos projetos anteriores para correções.

O processo de projeto pode envolver o desenvolvimento de vários modelos do sistema em diferentes níveis de abstração. Como um projeto é decomposto, os erros e as omissões em estágios anteriores são descobertos. Essas informações de feedback permitem o aprimoramento de modelos anteriores de projeto. A Figura 4.7 é um modelo desse processo, mostrando as descrições de projeto que devem ser produzidas nos diversos estágios. Esse diagrama sugere que os estágios do processo de projeto sejam seqüenciais. Na verdade, as atividades do processo de projeto são intercaladas. A realimentação de um estágio para outro e o conseqüente retrabalho do projeto são inevitáveis em todos os processos de projeto.

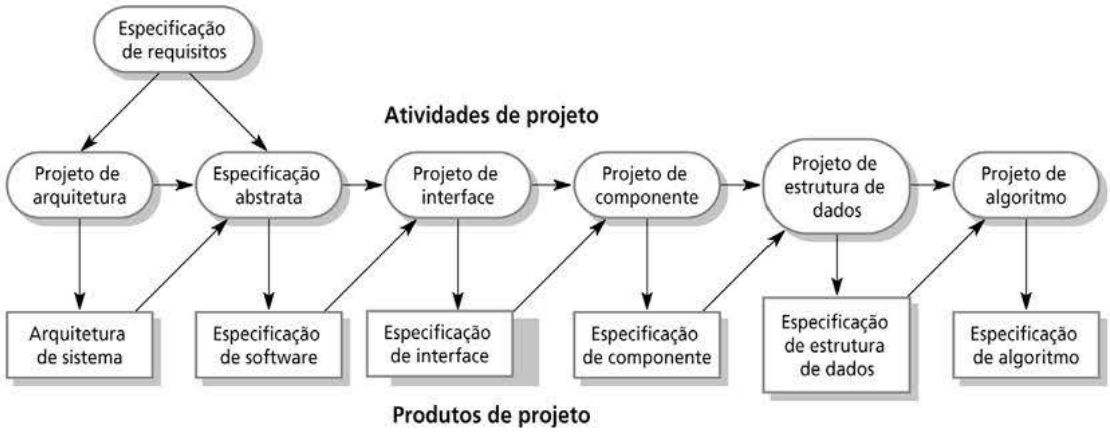


Figura 4.7
Modelo geral de processo de projeto.

A saída de cada atividade de projeto é uma especificação para o próximo estágio. Essa especificação pode ser uma especificação abstrata e formal produzida para esclarecer os requisitos, ou pode ser uma especificação de como uma parte do software será realizada. À medida que o processo de projeto prossegue, essas especificações tornam-se mais detalhadas. Os resultados finais do processo são especificações precisas de algoritmos e estruturas de dados a serem implementados.

As atividades específicas do processo de projeto são:

1. *Projeto de arquitetura*. Os subsistemas constituintes do sistema e os seus relacionamentos são identificados e documentados. Esse importante tópico é abordado nos capítulos 11, 12 e 13.
2. *Especificação abstrata*. Para cada subsistema, são produzidas uma especificação abstrata dos serviços e as restrições sob as quais ele deve operar.
3. *Projeto de interface*. Para cada subsistema, é projetada e documentada a interface com outros subsistemas. A especificação de interface não deve ser ambígua, pois ela permite que o subsistema seja usado sem que se tenha conhecimento da sua operação. Os métodos de especificação formal, discutidos no Capítulo 10, podem ser usados nesse estágio.
4. *Projeto de componente*. Os serviços são alocados aos componentes e as interfaces desses componentes são projetadas.
5. *Projeto de estruturas de dados*. As estruturas de dados usadas na implementação do sistema são projetadas detalhadamente e especificadas.
6. *Projeto de algoritmo*. Os algoritmos usados para fornecer os serviços são projetados detalhadamente e especificados.

Esse é um modelo geral do processo de projeto e os processos reais e práticos podem adaptá-lo de diferentes formas. As possíveis adaptações são:

1. Os últimos dois estágios de projeto — projeto de estruturas de dados e algoritmos — podem ser postergados até o processo de implementação.
2. Se uma abordagem exploratória de projeto for usada, as interfaces de sistema podem ser projetadas após a especificação das estruturas de dados.
3. O estágio de especificação abstrata pode ser omitido, embora seja uma parte essencial do projeto de sistemas críticos.

Cada vez mais, quando os métodos ágeis de desenvolvimento são usados (veja o Capítulo 17), as saídas do processo de projeto não serão documentos de especificação separados, mas sim representadas no código do programa. Após o projeto de arquitetura do sistema, os estágios posteriores do projeto são incrementais. Cada incremento é representado como código de programa, em vez de um modelo de projeto.

Uma abordagem diferente é feita pelos *métodos estruturados* para projeto, que contam com a produção de modelos gráficos de sistema (veja o Capítulo 8) e, em muitos casos, com a geração de código a partir desses modelos. Os métodos estruturados foram estabelecidos nos anos 1970, para apoiar o projeto orientado a funções (Constantine e Yourdon, 1979; Gane e Sarson, 1979). Foram propostos vários métodos concorrentes para apoiar o projeto orientado a objetos (Robinson, 1992; Booch, 1994) e eles foram unificados nos anos 1990 para criar a Unified Modeling Language (UML) e o processo de projeto associado (Rumbaugh, et al., 1991; Booch, et al., 1999; Rumbaugh, et al., 1999a; Rumbaugh, et al., 1999b). Enquanto estou escrevendo este texto, uma revisão ampla da UML (UML 2.0) está sendo feita.

Um método estruturado inclui um modelo de processo de projeto, notações para representar o projeto, formatos de relatórios, regras e recomendações de projeto. Os métodos estruturados podem apoiar alguns ou todos os modelos de um sistema:

1. Um modelo de objeto que apresenta as classes usadas no sistema e suas dependências.
2. Um modelo de seqüência que apresenta como os objetos do sistema interagem, quando o sistema está em execução.
3. Um modelo de transição de estado que mostra os estados de sistema e os disparadores das transições de um estado para outro.
4. Um modelo estruturado, no qual os componentes de sistema e as suas agregações são documentados.
5. Um modelo de fluxo de dados, no qual o sistema é modelado usando as transformações de dados que ocorrem durante sua execução. Esse modelo não é usado normalmente em métodos orientados a objetos, mas ainda é freqüentemente usado em projeto de sistemas de tempo real e de negócios.

Na prática, os 'métodos' estruturados são realmente notações padronizadas e incorporações de boas práticas. Seguindo esses métodos e aplicando as recomendações, pode-se chegar a um projeto razoável. A criatividade do projetista é ainda exigida para decidir sobre a decomposição do sistema e para garantir que o projeto capte adequadamente a especificação do sistema. Estudos empíricos de projetistas (Bansler e Bødker, 1993) mostraram que eles raramente seguem os métodos estritamente. Eles escolhem e selecionam as diretrizes de acordo com as circunstâncias locais.

O desenvolvimento de um programa para implementar o sistema parte naturalmente dos processos de projeto do sistema. Embora algumas classes de programas, como sistemas críticos de segurança, sejam geralmente projetadas detalhadamente antes do início de qualquer implementação, é mais comum que os estágios posteriores de projeto e desenvolvimento de programas sejam intercalados. As ferramentas CASE podem ser usadas para gerar um esqueleto de programa com base no projeto. Isso inclui código para definir e implementar interfaces e, em vários casos, o desenvolvedor precisa apenas adicionar detalhes da operação de cada componente do programa.

A programação é uma atividade pessoal e normalmente não existe um processo geral a seguir. Alguns programadores começarão com componentes que compreendem bem e irão desenvolvê-los para, então, passarem para aqueles mais mal compreendidos. Outros preferem a abordagem oposta, deixando os componentes mais familiares para o fim, pois eles sabem como desenvolvê-los. Alguns desenvolvedores gostam de definir os dados inicialmente no processo e, depois, usá-los para direcionar o desenvolvimento de programa; outros deixam os dados sem especificação por tanto tempo quanto for possível.

Normalmente, os programadores realizam algum teste de código desenvolvido por eles. Isso freqüentemente revela os defeitos de programa a serem removidos. Isso é chamado de *debugging*. O teste de defeito e o debugging são processos diferentes. O teste verifica a existência de defeitos. O debugging está relacionado à localização e correção desses defeitos.

A Figura 4.8 ilustra os estágios de debugging. Os defeitos no código devem ser localizados e o programa deve ser modificado para atender aos requisitos. O teste deve, então, ser repetido para garantir que a alteração foi feita corretamente. Dessa forma, o processo de depuração é parte tanto do desenvolvimento de software quanto do seu teste.

Durante o debugging, são geradas hipóteses sobre o comportamento observável do programa e essas hipóteses são testadas na esperança de encontrar o defeito que causou a anormalidade na saída. O teste de hipóteses pode envolver o rastreamento manual do código. Você pode escrever novos casos de teste para localizar o problema. Podem ser usadas ferramentas interativas de debugging, que mostram os valores intermediários de variáveis de programa e o rastreamento dos comandos executados, para auxiliar o processo de debugging.

4.3.3 Validação de software

A validação de software ou, mais genericamente, verificação e validação (V&V) destina-se a mostrar que um sistema está em conformidade com sua especificação e que atende às expectativas do cliente que está adquirindo o sistema. Isso envolve processos de verificação, tais como inspeções e revisões (veja o Capítulo 22), a cada estágio do processo de software, desde a definição de requisitos de usuário até o desenvolvimento do programa. A maior parte dos custos de validação, no entanto, incorrem após a implementação, quando o sistema operacional é testado (Capítulo 23).

Com exceção de programas pequenos, os sistemas não devem ser testados como uma unidade simples e monolítica. A Figura 4.9 mostra um processo de teste em três estágios, no qual os componentes do sistema são testados, depois é testado o sistema integrado e, finalmente, o sistema é testado com os dados do cliente. De maneira ideal, os defeitos de componentes são descobertos no início do processo e os problemas de interface, quando o sistema for integrado. No entanto, à medida que os defeitos são descobertos, o programa deve ser depurado e isso pode requerer que outros estágios no processo de teste sejam repetidos. Os erros nos componentes de programa podem aparecer durante o teste de sistema. O processo é, portanto, iterativo, com as informações sendo realimentadas dos estágios posteriores para as partes iniciais do processo.

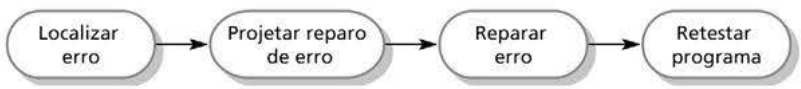
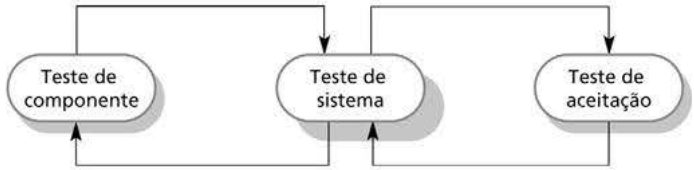


Figura 4.8
Processo de debugging.

Figura 4.9
Processo de teste.



Os estágios do processo de teste são:

1. *Teste de componente (ou unidade).* Os componentes individuais são testados para garantir que operem corretamente. Cada componente é testado independentemente, sem os outros componentes de sistema. Os componentes podem ser entidades simples, tais como funções ou classes de objetos, ou podem ser grupos coerentes dessas entidades.
2. *Teste de sistema.* Os componentes são integrados para compor o sistema. Esse processo está relacionado com a busca de erros que resultam das interações não previstas entre os componentes e problemas de interface de componentes. Está também relacionado à validação de que o sistema atende aos requisitos funcionais e não funcionais e ao teste de propriedades emergentes do sistema. Para sistemas de grande porte, isso pode ser um processo de vários estágios no qual os componentes são integrados para formar subsistemas testados individualmente antes que sejam integrados para formar o sistema final.
3. *Teste de aceitação.* Este é o estágio final do processo de teste, antes que o sistema seja aceito para o uso operacional. O sistema é testado com os dados fornecidos pelo cliente do sistema, em vez de dados simulados de teste. O teste de aceitação pode revelar erros e omissões na definição de requisitos de sistema, pois os dados reais exercitam o sistema de formas diferentes dos dados de teste. O teste de aceitação pode também revelar problemas de requisitos, em que os recursos do sistema não atendem realmente às necessidades do usuário ou o desempenho do sistema é inaceitável.

Geralmente, o desenvolvimento e o teste de componentes são intercalados. Os programadores elaboram seus próprios dados de teste e testam os códigos de forma incremental, à medida que são desenvolvidos. Essa é uma abordagem sensível economicamente, pois o programador conhece melhor o componente e é, portanto, a melhor pessoa para gerar os casos de teste.

Se for usada uma abordagem incremental para o desenvolvimento, cada incremento deve ser testado à medida que é desenvolvido, com testes baseados nos requisitos desse incremento. Na *extreme programming*, os testes são desenvolvidos junto com os requisitos, antes do início do desenvolvimento. Isso ajuda os testadores e os projetistas a compreender os requisitos e garante que não existam atrasos quando os casos de teste forem criados.

Os estágios posteriores de teste envolvem o trabalho de integração de vários programadores e devem ser planejados com antecedência. Uma equipe independente de testadores deve trabalhar nos planos de teste predefinidos, desenvolvidos baseados na especificação e projeto do sistema. A Figura 4.10 ilustra como os planos de teste são ligados entre as atividades de teste e de desenvolvimento.

O teste de aceitação é algumas vezes chamado de *teste alfa*. Os sistemas sob encomenda são desenvolvidos para um único cliente. O processo de teste alfa continua até que o projetista do sistema e o cliente concordem que o sistema liberado é uma implementação aceitável dos requisitos do sistema.

Quando um sistema será comercializado como um produto de software, freqüentemente é usado um processo de teste denominado *teste beta*. O teste beta envolve a liberação do sistema a uma série de clientes potenciais que concordam em usar esse sistema. Eles reportam os problemas para os desenvolvedores do sistema. Esse processo expõe o produto para

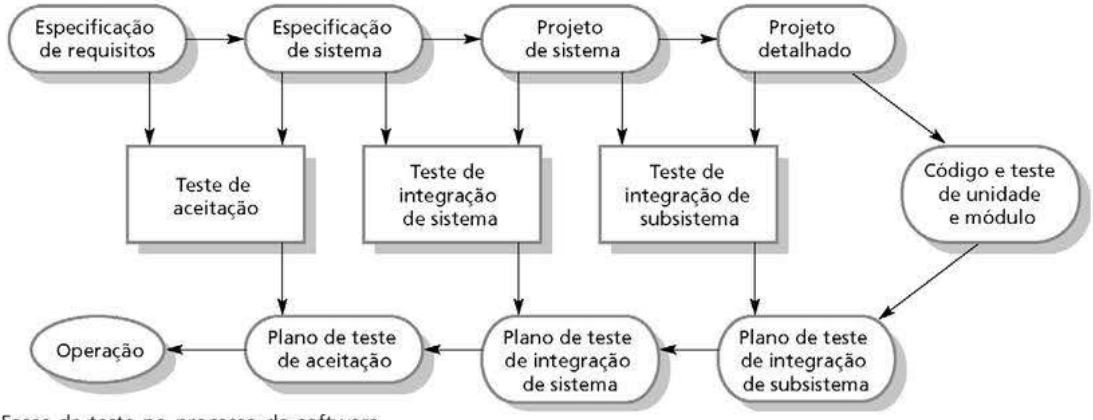


Figura 4.10 Fases de teste no processo de software.

uso real e detecta erros que podem não ter sido encontrados pelos criadores do sistema. Após este feedback, o sistema é modificado e liberado para o próximo teste beta ou para venda normal.

4.3.4 Evolução de software

A flexibilidade dos sistemas de software é uma das principais razões pelas quais cada vez mais o software está sendo incorporado a sistemas complexos de grande porte. Após a decisão de aquisição de hardware, é muito oneroso fazer mudanças no projeto de hardware. No entanto, mudanças podem ser feitas no software a qualquer instante durante ou após o desenvolvimento do sistema. Mesmo mudanças extensas são mais baratas do que as mudanças correspondentes no hardware do sistema.

Historicamente, sempre existiu uma separação entre o processo de desenvolvimento de software e o processo de evolução de software (manutenção de software). As pessoas pensam no processo de desenvolvimento de software como uma atividade criativa, em que o sistema é desenvolvido a partir de um conceito inicial até um sistema funcional. No entanto, elas pensam na manutenção de software como algo enfadonho e desinteressante. Embora os custos de 'manutenção' freqüentemente sejam várias vezes maiores que os custos iniciais de desenvolvimento, os processos de manutenção são, às vezes, considerados como menos desafiadores do que o desenvolvimento original de software.

A distinção entre o desenvolvimento e a manutenção tem se tornado cada vez mais irrelevante. Atualmente, poucos sistemas de software são completamente novos e faz muito mais sentido considerar o desenvolvimento e a manutenção de forma contínua. Em vez de dois processos separados, é mais realista pensar na engenharia de software como um processo evolutivo (Figura 4.11), no qual o software é continuamente alterado ao longo de sua vida, em resposta às mudanças de requisitos e às necessidades do cliente.

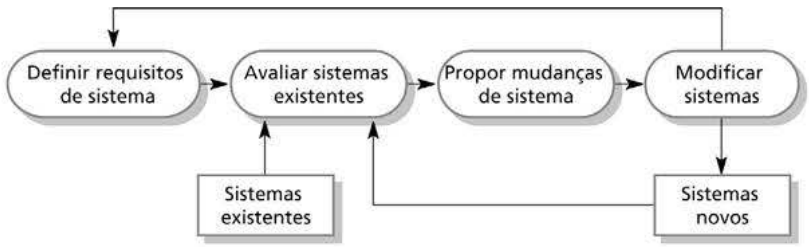


Figura 4.11
Evolução de sistema.

4.4 O Rational Unified Process

O Rational Unified Process (RUP) é um exemplo de modelo de processo moderno que foi derivado do trabalho sobre a UML e do Processo Unificado de Desenvolvimento de Software associado (Rumbaugh, et al., 1999b). Incluí uma descrição nesta seção por ser um bom exemplo de modelo híbrido de processo. Ele traz elementos de todos os modelos genéricos de processo (Seção 4.1), apóia a iteração (Seção 4.2) e ilustra boas práticas de especificação e projeto (Seção 4.3).

O RUP reconhece que os modelos convencionais de processo apresentam uma visão única de processo. Por outro lado, o RUP é geralmente descrito a partir de três perspectivas:

1. Uma perspectiva dinâmica, que mostra as fases do modelo ao longo do tempo.
2. Uma perspectiva estática, que mostra as atividades realizadas no processo.
3. Uma perspectiva prática, que sugere as boas práticas a serem usadas durante o processo.

A maior parte das descrições do RUP tenta combinar as perspectivas estática e dinâmica em um único diagrama (Krutchen, 2000). Penso que isso torna o processo difícil de ser compreendido e, portanto, uso descrições separadas de cada uma dessas perspectivas.

O RUP é um modelo constituído por fases que identifica quatro fases discretas no processo de software. No entanto, ao contrário do modelo em cascata, no qual as fases coincidem com as atividades do processo, as fases do RUP estão relacionadas mais estritamente aos negócios do que a assuntos técnicos. A Figura 4.12 mostra as fases do RUP. São elas:

1. *Concepção.* O objetivo da fase de concepção é estabelecer um *business case* para o sistema. Você deve identificar todas as entidades externas (pessoas e sistemas), que irão interagir com o sistema, e definir essas interações. Depois você usa essas informações para avaliar a contribuição do sistema com o negócio. Se a contribuição for de pouca importância, o projeto pode ser cancelado depois dessa fase.

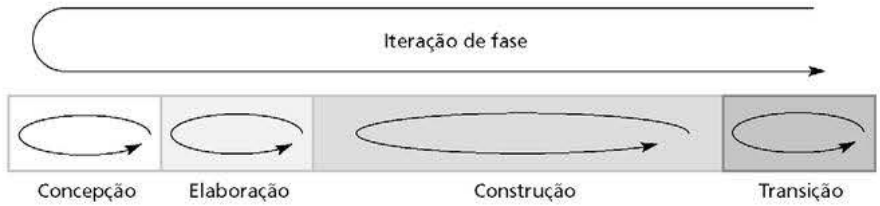


Figura 4.12
Fases no Rational Unified Process.

2. *Elaboração.* Os objetivos da fase de elaboração são desenvolver um entendimento do domínio do problema, estabelecer um framework de arquitetura para o sistema, desenvolver o plano de projeto e identificar os riscos principais do projeto. Ao concluir essa fase, você deve ter um modelo de requisitos para o sistema (os casos de uso da UML são especificados), uma descrição de arquitetura e um plano de desenvolvimento para o software.
3. *Construção.* A fase de construção está essencialmente relacionada ao projeto, programação e teste de sistema. As partes do sistema são desenvolvidas paralelamente e integradas durante esta fase. Ao concluir esta fase, você deve ter um sistema de software em funcionamento e a documentação associada pronta para ser liberada para os usuários.
4. *Transição.* A fase final do RUP está relacionada à transferência do sistema da comunidade de desenvolvimento para a comunidade dos usuários e com a entrada do sistema em funcionamento no ambiente real. Isso é algo ignorado na maioria dos modelos de processo de software, mas é, de fato, uma atividade onerosa e, às vezes, problemática. Ao concluir esta fase, você deverá ter um sistema de software documentado, funcionando corretamente em seu ambiente operacional.

A iteração no RUP é considerada em duas formas, como apresentado na Figura 4.12. Cada fase pode ser realizada de forma iterativa, com os resultados desenvolvidos incrementalmente. Além disso, o conjunto total de fases pode também ser realizado de forma incremental, conforme apresentado pela seta retornando da Transição para a Concepção na Figura 4.12.

A visão estática do RUP enfoca as atividades que ocorrem durante o processo de desenvolvimento. Elas são chamadas de *workflows* na descrição do RUP. Existem seis workflows de processo principais identificados e três workflows de apoio principais. O RUP foi projetado em conjunto com a UML — uma linguagem de modelagem orientada a objetos — e, por isso, a descrição dos workflows é orientada em termos dos modelos UML associados. Os principais workflows de engenharia e de apoio estão descritos na Tabela 4.1.

A vantagem de apresentar as visões estática e dinâmica é que as fases do processo de desenvolvimento não estão associadas aos workflows específicos. Em princípio, todos os workflows do RUP podem ser ativados em todos os estágios do processo. Naturalmente, a maior parte do esforço será provavelmente despendida nos workflows como modelagem de negócios e requisitos nas fases iniciais do processo e no teste e implantação nas fases posteriores.

Tabela 4.1 Workflows estáticos no Rational Unified Process.

Workflow	Descrição
Modelagem de negócios	Os processos de negócios são modelados usando casos de uso de negócios.
Requisitos	Os agentes que interagem com o sistema são identificados e os casos de uso são desenvolvidos para modelar os requisitos de sistema.
Análise e projeto	Um modelo de projeto é criado e documentado usando modelos de arquitetura, modelos de componente, modelos de objeto e modelos de seqüência.
Implementação	Os componentes de sistema são implementados e estruturados em subsistemas de implementação. A geração automática de código com base nos modelos de projeto ajuda a acelerar esse processo.
Teste	O teste é um processo iterativo realizado em conjunto com a implementação. O teste de sistema segue o término da implementação.
Implantação	Uma versão do produto é criada, distribuída aos usuários e instalada no local de trabalho.
Gerenciamento de configuração e mudanças	Este workflow de apoio gerencia as mudanças do sistema (veja o Capítulo 29).
Gerenciamento de projetos	Este workflow de apoio gerencia o desenvolvimento do sistema (veja o Capítulo 5).
Ambiente	Este workflow está relacionado à disponibilização de ferramentas apropriadas de software para a equipe de desenvolvimento.

A perspectiva prática do RUP descreve boas práticas de engenharia de software recomendadas para uso em desenvolvimento de sistemas. São recomendadas seis melhores práticas fundamentais:

1. *Desenvolver o software iterativamente.* Planejar os incrementos de software com base nas prioridades do cliente e desenvolver e entregar antes as características de sistema de maior prioridade no processo de desenvolvimento.
2. *Gerenciar requisitos.* Documentar explicitamente os requisitos do cliente e manter acompanhamento das mudanças desses requisitos. Analisar o impacto das mudanças sobre o sistema antes de aceitá-las.
3. *Usar arquiteturas baseadas em componentes.* Estruturar a arquitetura de sistema com componentes, conforme explicado anteriormente neste capítulo.
4. *Modelar o software visualmente.* Usar modelos gráficos de UML para apresentar as visões estática e dinâmica do software.
5. *Verificar a qualidade do software.* Garantir que o software atenda aos padrões de qualidade da organização.
6. *Controlar as mudanças do software.* Gerenciar as mudanças do software, usando um sistema de gerenciamento de mudanças e procedimentos e ferramentas de gerenciamento de configuração (veja o Capítulo 29).

O RUP não é um processo adequado a todos os tipos de desenvolvimento, mas representa uma nova geração de processos genéricos. A mais importante inovação é a separação de fases e workflows, e o reconhecimento de que a implantação de software no ambiente do usuário é parte do processo. As fases são dinâmicas e têm objetivos. Os workflows são estáticos e constituem atividades técnicas que não estão associadas a uma única fase, mas podem ser utilizados ao longo do desenvolvimento para atingir os objetivos de cada fase.

4.5 Engenharia de Software Auxiliada por Computador

A Engenharia de Software Auxiliada por Computador (CASE — *Computer-Aided Software Engineering*) é o nome dado ao software usado para apoiar as atividades de processo de software, como engenharia de requisitos, projeto, desenvolvimento de programas e teste. As ferramentas CASE, portanto, incluem editores de diagramas, dicionário de dados, compiladores, debuggers, ferramentas de construção de sistemas etc.

A tecnologia CASE fornece apoio ao processo de software pela automação de algumas atividades de processo e pelo fornecimento de informações sobre o software que está sendo desenvolvido. Exemplos de atividades que podem ser automatizadas com o uso de CASE incluem:

1. O desenvolvimento dos modelos gráficos de sistema como parte da especificação de requisitos ou do projeto de software.
2. A compreensão de um projeto por intermédio do uso de um dicionário de dados com informações sobre as entidades e as relações em um projeto.
3. A geração de interfaces com o usuário com base em uma descrição de interface gráfica criada interativamente pelo usuário.
4. O debugging do programa por meio do fornecimento de informações sobre um programa em execução.
5. A tradução automática de programas a partir de uma versão antiga de uma linguagem de programação, como COBOL, para uma versão mais recente.

A tecnologia CASE está disponível atualmente para a maioria das atividades rotineiras no processo de software. Isso levou a alguns aprimoramentos de qualidade e produtividade de software, embora estes sejam menores do que os previstos pelos primeiros defensores de CASE. Esses defensores sugeriram que provavelmente os aprimoramentos seriam enormes caso fossem usados os ambientes integrados de CASE. Na verdade, os aprimoramentos atingidos são da ordem de 40% (Huff, 1992). Embora isso seja significativo, as previsões, na ocasião da introdução de ferramentas CASE nas décadas de 1980 e 1990, eram de que o uso da tecnologia CASE geraria uma imensa redução nos custos de processo de software.

Os aprimoramentos gerados pelo uso de CASE são limitados por dois fatores:

1. A engenharia de software é, essencialmente, uma atividade de projeto, baseada no pensamento criativo. Os sistemas CASE existentes automatizam as atividades rotineiras, mas as tentativas de aproveitar a tecnologia de inteligência artificial para apoiar o projeto não foram bem-sucedidas.
2. Na maioria das organizações, a engenharia de software é uma atividade de equipe e os engenheiros de software passam uma grande parte do tempo interagindo com outros membros da equipe. A tecnologia CASE não pode ajudar muito nesse caso.

A tecnologia CASE atingiu a maturidade e ferramentas CASE e workbenches estão disponíveis em uma grande gama de fornecedores. No entanto, em vez de enfocar uma ferramenta específica, apresento nesta seção simplesmente uma visão geral de ferramentas, com alguma explicação sobre apoio específico em outros capítulos. Em minhas páginas Web, incluí links para material adicional sobre CASE e para fornecedores de ferramentas CASE.

4.5.1 Classificação CASE

As classificações CASE ajudam a compreender os tipos das ferramentas CASE e seu papel em apoiar as atividades do processo de software. Existem várias formas de classificar as ferramentas CASE. Cada uma delas fornece uma perspectiva diferente sobre essas ferramentas. Nesta seção, explico as ferramentas CASE a partir de três dessas perspectivas:

1. *Uma perspectiva funcional,* em que as ferramentas CASE são classificadas de acordo com sua função específica.
2. *Uma perspectiva de processo,* em que as ferramentas são classificadas de acordo com as atividades de apoio que fornecem.
3. *Uma perspectiva de integração,* em que as ferramentas CASE são classificadas de acordo com sua organização em unidades integradas que apóiam uma ou mais atividades de processo.

A Tabela 4.2 é uma classificação de ferramentas CASE de acordo com a função. Essa tabela lista uma quantidade de tipos diferentes de ferramentas CASE e apresenta exemplos de cada uma. Não é uma lista completa de ferramentas CASE. As ferramentas especializadas, tais como ferramentas para apoio ao reuso, não foram incluídas.

A Figura 4.13 apresenta uma classificação alternativa de ferramentas CASE. Mostra as fases de processo apoiadas por vários tipos de ferramentas CASE. As ferramentas para planejamento e estimativa, edição de texto, preparação de documentos e gerenciamento de configuração podem ser usadas ao longo do processo de software.

A abrangência de apoio ao processo de software oferecido pela tecnologia CASE é uma outra dimensão possível de classificação. Fuggetta (Fuggetta, 1993) propôs que os sistemas CASE devem ser classificados em três categorias:

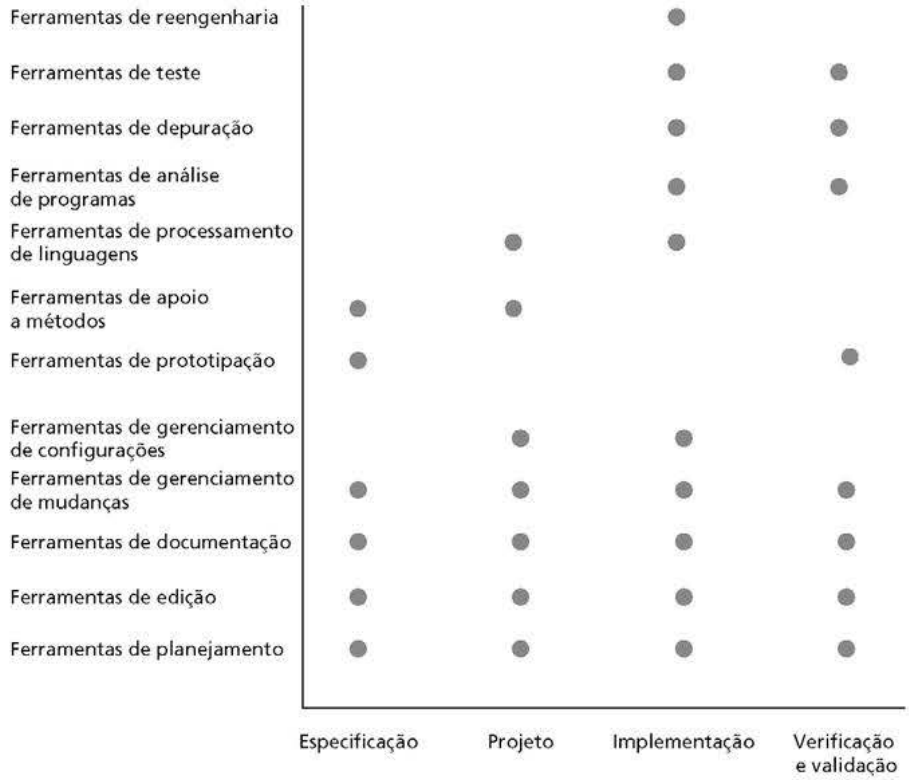
1. *Ferramentas* apóiam tarefas individuais de processo, como verificação de consistência de um projeto, compilação de um programa e comparação de resultados de teste. As ferramentas podem ser de propósito geral e independentes (por exemplo, um processador de texto) ou agrupadas em workbenches.
2. *Workbenches* apóiam as fases ou atividades de processo, como especificação, projeto etc. Eles consistem geralmente em um conjunto de ferramentas com integração de maior ou menor nível.
3. *Ambientes* apóiam todo ou pelo menos uma parte substancial do processo de software. Eles normalmente incluem vários *workbenches* integrados.

Tabela 4.2 Classificação funcional de ferramentas CASE.

Tipo de ferramenta	Exemplos
Ferramentas de planejamento	Ferramentas PERT, ferramentas para estimativas, planilhas
Ferramentas de edição	Editores de texto, editores de diagramas, processadores de texto
Ferramentas de gerenciamento de mudanças	Ferramentas de controle de requisitos, sistemas de controle de mudanças
Ferramentas de gerenciamento de configuração	Sistemas de gerenciamento de versões, ferramentas de construção de sistemas
Ferramentas de prototipação	Linguagens de nível muito alto, geradores de interface com o usuário
Ferramentas de apoio a métodos	Editores de projeto, dicionários de dados, geradores de código
Ferramentas de processamento de linguagens	Compiladores, interpretadores
Ferramentas de análise de programa	Geradores de referências cruzadas, analisadores estáticos, analisadores dinâmicos
Ferramentas de teste	Geradores de dados de teste, comparadores de arquivos
Ferramentas de depuração	Sistemas de depuração interativos
Ferramentas de documentação	Programas de formatação de páginas, editores de imagens
Ferramentas de reengenharia	Sistemas de referência cruzada, sistemas de reestruturação de programas

Figura 4.13

Classificação de ferramentas CASE baseada em atividades.

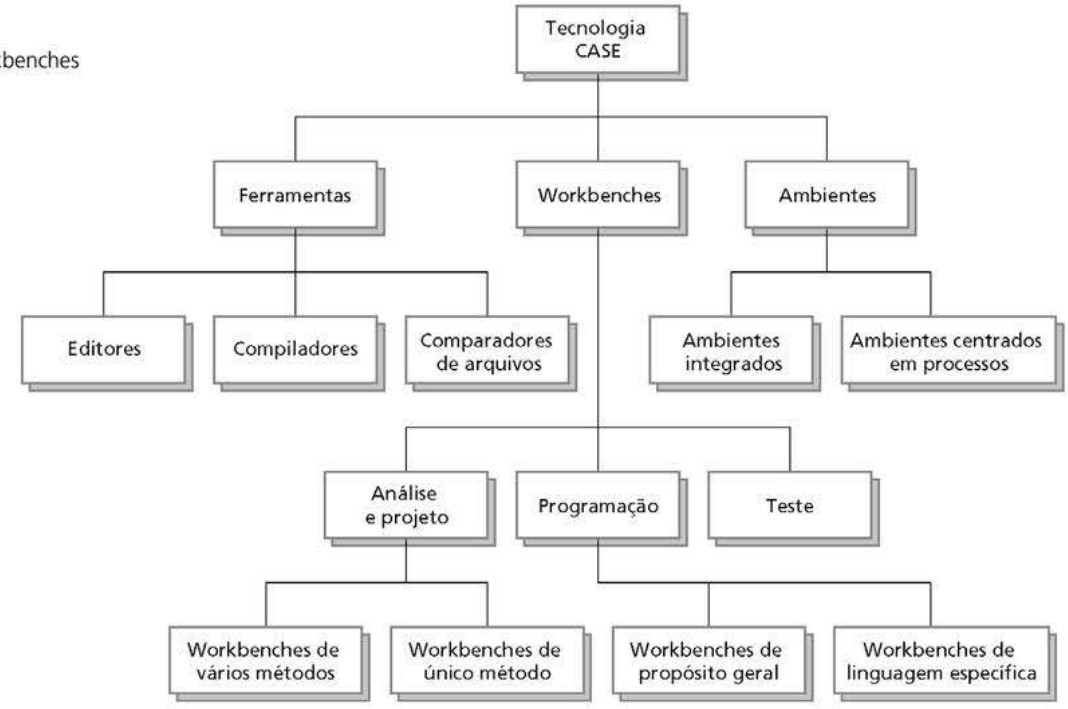


A Figura 4.14 ilustra essa classificação e mostra alguns exemplos dessas classes de apoio de CASE. Naturalmente, é um exemplo ilustrativo; vários tipos de ferramentas e workbenches foram deixados fora desse diagrama.

As ferramentas de propósito geral são usadas a critério do engenheiro de software que toma decisões sobre quando aplicá-las para apoio ao processo. Os workbenches, no entanto, geralmente apóiam algum método que inclui um modelo de processo e um conjunto de regras/diretrizes aplicadas ao software em desenvolvimento. Classifiquei os ambientes como integrados ou centrados em processos. Os ambientes integrados fornecem apoio de infra-estrutura para a integração, controle

Figura 4.14

Ferramentas, workbenches e ambientes.



e apresentação de dados. Os ambientes centrados em processos são mais genéricos. Eles incluem o conhecimento de processo e um mecanismo de processo que usa esse modelo de processo para orientar os engenheiros sobre quais ferramentas ou workbenches aplicar e quando eles devem ser usados.

Na prática, o limite entre essas classes é tênue. As ferramentas podem ser vendidas como um único produto, mas podem oferecer apoio a diferentes atividades. Por exemplo, a maioria dos processadores de texto oferece um editor de diagramas incorporado. Os workbenches CASE para projeto geralmente apóiam a programação e teste e, assim, são mais parecidos com ambientes do que workbenches especializados. Portanto, pode não ser sempre fácil incluir um produto em uma classificação. Contudo, a classificação fornece um primeiro passo útil para ajudar a compreender a extensão do apoio ao processo que a ferramenta fornece.

PONTOS-CHAVE

- Processos de software são atividades envolvidas na produção de um sistema de software. Os modelos de processo de software são representações abstratas desses processos.
- Todos os processos de software incluem especificação, projeto e implementação, validação e evolução de software.
- Modelos genéricos de processo descrevem a organização de processos de software. Exemplos de modelos genéricos incluem o modelo em cascata, desenvolvimento evolucionário e engenharia de software baseada em componentes.
- Modelos de processo iterativos apresentam o processo de software como um ciclo de atividades. A vantagem dessa abordagem é que ela evita um comprometimento prematuro com uma especificação ou projeto. Exemplos de modelos iterativos incluem desenvolvimento incremental e o modelo espiral.
- A engenharia de requisitos é o processo de desenvolvimento de uma especificação de software. As especificações são usadas para comunicar as necessidades de sistema do cliente a seus arquitetos.
- Os processos de projeto e implementação concentram-se na transformação de uma especificação de requisitos em um sistema de software executável. Métodos sistemáticos de projeto podem ser usados como parte dessa transformação.
- A validação de software é o processo que verifica se o sistema está em conformidade com as especificações e se atende às reais necessidades dos usuários do sistema.
- A evolução de software concentra-se nas modificações existentes nos sistemas de software para atender aos novos requisitos. Essa abordagem está se tornando comum no desenvolvimento de software de sistemas de pequeno e médio portes.
- O Rational Unified Process é um modelo de processo genérico e moderno, organizado em fases (concepção, elaboração, construção e transição), mas que separa as atividades (requisitos, análise e projeto etc.) dessas fases.
- A tecnologia CASE fornece um apoio automatizado aos processos de software. As ferramentas CASE apóiam as atividades individuais de processo; os workbenches apóiam um conjunto de atividades relacionadas; os ambientes apóiam todas ou a maioria das atividades de processo.

LEITURAS SUGERIDAS

Extreme programming explained: embrace change. Um livro que defende e explica o processo e as experiências de *extreme programming*. O autor foi o inventor da *extreme programming* e deixa claro o seu entusiasmo. (Kent Beck, 2000, Addison-Wesley.)

The rational unified process — an introduction. Este era o livro mais legível disponível sobre RUP no momento em que este livro estava sendo escrito. Krutchen descreve bem o processo, mas eu gostaria de saber mais sobre as dificuldades práticas do uso do processo. (P. Krutchen, 2000, Addison-Wesley.)

Managing software quality and business risk. Este livro trata principalmente do gerenciamento de software, mas inclui um excelente capítulo (Capítulo 4) sobre modelos de processo. (M. Ould, 1999, John Wiley & Sons.)

"A classification of CASE technology". O esquema de classificação proposto neste artigo é utilizado neste capítulo, mas Fuggetta inclui mais detalhes e ilustra como vários produtos comerciais se enquadram nesse esquema. (A. Fuggetta, *IEEE computer*, 26 (12), dezembro de 1993.)

EXERCÍCIOS

4.1 Justificando sua resposta com base no tipo de sistema a ser desenvolvido, sugira o modelo genérico de processo de software mais apropriado que poderia ser usado como base para gerenciar o desenvolvimento dos seguintes sistemas:

- Um sistema para controlar um antibloqueador de freios em um automóvel
- Um sistema de realidade virtual para apoiar a manutenção de software

- Um sistema de contabilidade de universidade que substitui um sistema existente
- Um sistema interativo que permite aos passageiros encontrar o horário dos trens por meio de terminais instalados nas estações.

4.2 Explique por que os programas desenvolvidos por desenvolvimento evolucionário são provavelmente difíceis de serem mantidos.

4.3 Explique como o modelo em cascata de processo de software e o modelo de prototipação podem ser acomodados no modelo de processo espiral.

4.4 Quais são as vantagens de fornecer as visões estática e dinâmica do processo de software como no Rational Unified Process?

4.5 Sugira por que é importante fazer uma distinção entre o desenvolvimento dos requisitos de usuário e os requisitos de sistema no processo de engenharia de requisitos.

4.6 Descreva as principais atividades no processo de projeto de software e as saídas dessas atividades. Usando um diagrama, mostre os possíveis relacionamentos entre as saídas dessas atividades.

4.7 Quais são os cinco componentes de um método de projeto? Descreva os componentes de algum método conhecido. Avalie a abrangência do método escolhido.

4.8 Projete um modelo de processo para executar os testes de sistema e registrar seus resultados.

4.9 Explique por que um sistema de software usado em um ambiente real deve mudar sob pena de tornar-se progressivamente menos útil.

4.10 Sugira como um esquema de classificação da tecnologia CASE pode ser útil para gerentes responsáveis pela aquisição de sistema CASE.

4.11 Faça um levantamento da disponibilidade de ferramentas em seu ambiente local de desenvolvimento e classifique as ferramentas de acordo com os parâmetros (função, atividade, abrangência de apoio) sugeridos neste livro.

4.12 Historicamente, a introdução de tecnologia causou profundas mudanças no mercado de trabalho e, pelo menos temporariamente, deixou as pessoas desempregadas. Explique se a introdução de tecnologia avançada de CASE apresenta probabilidade de causar as mesmas conseqüências para os engenheiros de software. Se você não achar que isso é possível, explique por quê. Se achar que isso reduzirá as oportunidades de trabalho, é ético que os engenheiros afetados resistam passiva ou ativamente à introdução dessa tecnologia?

5

Gerenciamento de projetos

Objetivos

O objetivo deste capítulo é proporcionar uma visão geral do gerenciamento de projetos de software. Depois de ler este capítulo, você:

- conhecerá as principais tarefas dos gerentes de projeto de software;
- compreenderá por que a natureza do software torna o gerenciamento de projetos de software mais difícil do que o gerenciamento de projetos de outros tipos de engenharia;
- descobrirá a necessidade do planejamento de projetos em todos os projetos de software;
- saberá como as representações gráficas (diagramas de barras e diagramas de atividades) podem ser usadas pelos gerentes de projeto para representar os cronogramas de projeto;
- terá sido apresentado à noção de gerenciamento de riscos e alguns dos riscos que podem surgir em projetos de software.

Conteúdo

5.1 Atividades de gerenciamento
5.2 Planejamento de projeto
5.3 Cronograma do projeto
5.4 Gerenciamento de riscos

O gerenciamento de projetos de software é uma parte essencial da engenharia de software. Um bom gerenciamento não pode garantir o sucesso de um projeto. No entanto, um mau gerenciamento geralmente resulta em falha do projeto: o software é entregue com atraso, custa mais do que foi originalmente estimado e falha ao atender seus requisitos.

Os gerentes de software são responsáveis pelo desenvolvimento de planos e cronogramas do projeto. Eles supervisionam o trabalho para assegurar que ele esteja sendo realizado dentro dos padrões exigidos e monitoram o progresso para verificar se o desenvolvimento está no prazo e dentro do orçamento. O gerenciamento de projetos de software é necessário, pois a engenharia de software profissional está sempre sujeita às restrições de orçamento e de cronograma da organização. O trabalho do gerente de projeto de software é assegurar que o projeto de software atenda a essas restrições e entregue um software que contribua para as metas da empresa que está desenvolvendo o software.

Os gerentes de software fazem o mesmo tipo de trabalho que os gerentes de projeto de outras áreas da engenharia. No entanto, a engenharia de software é diferente de outros tipos de engenharia em uma série de maneiras. Essas distinções tornam o gerenciamento de software particularmente difícil. Algumas das diferenças são:

1. *O produto é intangível*. O gerente de um projeto de construção de um navio ou de um projeto de engenharia civil pode ver o produto que está sendo construído. Se o cronograma atrasa, o efeito sobre o produto é visível — partes da estrutura estarão, obviamente, não concluídas. O software é intangível. Não pode ser visto ou tocado. Os gerentes de projetos de software não podem ver o progresso. Eles contam com outras pessoas para produzir a documentação necessária para examinar o progresso.
2. *Não existem processos-padrão de software*. Em áreas da engenharia com um longo histórico, o processo é experimentado e testado. O processo de engenharia de alguns tipos de sistema, como pontes e prédios, é bem compreendido. No entanto, os processos de software variam drasticamente de uma organização para outra. Embora nossa compreensão desses processos tenha se desenvolvido de modo significativo nos últimos anos, ainda não podemos prever, confiavelmente, quando determinado processo de software provavelmente apresentará problemas de desenvolvimento. Isso é especialmente verdadeiro quando o projeto de software é parte de um projeto mais amplo de engenharia de sistemas.
3. *Projetos de software de grande porte são, freqüentemente, projetos 'únicos'*. Projetos de software de grande porte são geralmente diferentes, de alguma forma, de projetos anteriores. Portanto, mesmo os gerentes que têm uma grande experiência podem considerar difícil prever problemas. Além disso, as rápidas mudanças de tecnologia em computadores e comunicações podem tornar a experiência do gerente obsoleta. As lições aprendidas em projetos anteriores podem não ser aproveitadas em novos projetos.

Devido a esses problemas, não é de surpreender que alguns projetos de software atrasem, ultrapassem o orçamento e estejam fora do cronograma. Os sistemas de software são freqüentemente novos e tecnologicamente inovadores. Os projetos inovadores de engenharia (como novos sistemas de transporte), freqüentemente apresentam também problemas de cronograma. Em face das dificuldades envolvidas, é surpreendente que tantos projetos de software sejam entregues no prazo e dentro do orçamento!

O gerenciamento de projetos de software é um assunto amplo e não pode ser abordado apenas em um único capítulo. Portanto, apresento simplesmente uma introdução do assunto neste capítulo e descrevo três atividades importantes de gerenciamento: planejamento de projeto, desenvolvimento de cronograma de projeto e gerenciamento de riscos. Os capítulos posteriores (na Parte 6) abordarão outros aspectos do gerenciamento de software, incluindo gerenciamento de pessoas, estimativa de custos de software e gerenciamento de qualidade.

5.1 Atividades de gerenciamento

É impossível fornecer uma descrição de trabalho-padrão para um gerente de software. O trabalho varia muito, dependendo da organização e do produto de software que está sendo desenvolvido. No entanto, a maioria dos gerentes assume a responsabilidade, em algum estágio, por algumas ou todas as seguintes atividades:

- Elaboração da proposta
- Planejamento e desenvolvimento do cronograma do projeto
- Custo do projeto
- Monitoração e revisões do projeto
- Seleção e avaliação de pessoal
- Elaboração de relatórios e apresentações

O primeiro estágio de um projeto de software pode envolver a elaboração de uma proposta para obter um contrato para realizar o trabalho. A proposta descreve os objetivos do projeto e como ele será realizado. Geralmente inclui a estimativa de custos e de cronograma e justifica por que o contrato deve ser concedido a determinada organização ou equipe. A elaboração da proposta é uma tarefa crítica, pois a existência de muitas organizações de software depende da existência de um número suficiente de propostas aceitas e contratos concedidos. Pode não haver diretrizes definidas para essa tarefa; a elaboração da proposta é uma habilidade adquirida pela prática e pela experiência.

O planejamento de projeto está relacionado à identificação das atividades, marcos e produtos gerados por um projeto. É elaborado um plano para guiar o desenvolvimento em direção aos objetivos do projeto. A estimativa de custos é uma atividade relacionada à estimativa dos recursos necessários para realizar o plano do projeto. Abordo esses itens mais detalhadamente neste capítulo e no Capítulo 26.

A monitoração do projeto é uma atividade contínua. O gerente deve manter o acompanhamento do progresso do projeto e comparar o progresso com os custos reais e planejados. Embora a maioria das organizações possua mecanismos formais para monitoração, um gerente experiente pode fornecer um quadro mais nítido do que está acontecendo por meio de discussões informais com a equipe do projeto.

A monitoração informal pode prever problemas potenciais do projeto, revelando dificuldades à medida que elas ocorrem. Por exemplo, discussões diárias com a equipe do projeto podem revelar um problema específico ao localizar algum defeito de software. Em vez de aguardar que um atraso no cronograma seja relatado, o gerente de software poderia designar algum especialista para o problema ou pode propor uma alternativa de programação.

Durante um projeto, é normal que ocorram uma série de revisões formais de gerenciamento do projeto. Elas estão relacionadas à revisão geral do progresso e desenvolvimento técnico do projeto e à verificação se o projeto e as metas da organização que está financiando o software ainda estão alinhados.

O resultado de uma revisão pode levar à decisão de cancelamento de um projeto. O tempo de desenvolvimento de um projeto de software de grande porte pode ser de vários anos. Durante esse período, é quase certo que ocorram mudanças nos objetivos da organização. Essas mudanças podem significar que o software não é mais necessário ou que os requisitos originais de projeto não são apropriados. A gerência deve decidir pela interrupção do desenvolvimento do software ou alterar o projeto para acomodar as mudanças nos objetivos da organização.

Os gerentes geralmente precisam selecionar pessoas para trabalhar em seus projetos. O ideal é que haja o pessoal habilitado com experiência adequada disponível para trabalhar no projeto. No entanto, na maioria dos casos, os gerentes precisam aceitar uma equipe de projeto aquém do considerado ideal. As razões para isso são:

1. O orçamento do projeto pode não ser suficiente para contratar uma equipe muito bem remunerada. Pode ser necessário contratar uma equipe menos experiente e que aceite uma remuneração menor.
2. Uma equipe com experiência adequada pode não estar disponível na organização ou externamente. Pode ser impossível recrutar uma nova equipe para o projeto. Dentro da organização, as pessoas mais indicadas já podem estar alocadas em outros projetos.
3. A organização pode querer desenvolver as habilidades de seus funcionários. Uma equipe inexperiente pode ser designada a um projeto a fim de aprender e ganhar experiência.

O gerente de software precisa trabalhar dentro dessas restrições ao selecionar a equipe de projeto. Mas provavelmente ocorrerão problemas, a menos que pelo menos um membro do projeto tenha alguma experiência com o tipo do sistema a ser desenvolvido. Sem essa experiência, muitos erros simples poderão ser cometidos. Explico a montagem da equipe e a seleção do pessoal no Capítulo 25.

Os gerentes de projeto são geralmente responsáveis pela preparação de relatórios sobre o projeto para o cliente e para as organizações contratantes. Eles devem redigir documentos concisos e coerentes que resumam as informações fundamentais com base em relatórios detalhados do projeto. Devem estar aptos a apresentar essas informações durante as revisões do progresso. Conseqüentemente, se você for um gerente de projeto, deverá ser capaz de se comunicar eficientemente de forma verbal e escrita.

5.2 Planejamento de projeto

O gerenciamento eficiente de um projeto de software depende de um planejamento minucioso do progresso do projeto. Os gerentes devem prever os problemas que podem ocorrer e preparar soluções experimentais para esses problemas. Um plano elaborado no início de um projeto deve ser usado como guia. Esse plano inicial deve ser o melhor possível em face das informações disponíveis. Ele deve evoluir à medida que o projeto progride e melhores informações se tornem disponíveis.

A estrutura de um plano de desenvolvimento de software é descrita na Seção 5.2.1. Assim como um plano de projeto, os gerentes podem também precisar elaborar outros tipos de planos. Eles são descritos brevemente na Tabela 5.1 e abordados mais detalhadamente em um capítulo relevante em outra parte no livro.

O pseudocódigo apresentado na Figura 5.1 estabelece um processo de planejamento de projeto para desenvolvimento de software. Ele mostra que o planejamento é um processo iterativo, apenas concluído quando o próprio projeto é concluído. À medida que as informações se tornam disponíveis durante o projeto, o plano deve ser regularmente revisado. As metas da empresa constituem um importante fator que deve ser considerado na formulação do plano do projeto. À medida que essas metas mudam, as metas do projeto também mudam e, portanto, são necessárias mudanças no plano do projeto.

No início de um processo de planejamento, você deve avaliar as restrições (data de entrega definida, pessoal disponível, orçamento geral etc.) que afetam o projeto. Junto com isso, você deve estimar os parâmetros do projeto, como sua estrutura,

Tabela 5.1 Tipos de planos

Plano	Descrição
Plano de qualidade	Descreve os procedimentos e os padrões de qualidade usados no projeto. Veja o Capítulo 24.
Plano de validação	Descreve a abordagem, os recursos e o cronograma usados para a validação do sistema. Veja o Capítulo 19.
Plano de gerenciamento de configuração	Descreve os procedimentos e as estruturas de gerenciamento de configuração a serem usados. Veja o Capítulo 29.
Plano de manutenção	Prevê os requisitos de manutenção do sistema, os custos de manutenção e o esforço necessário. Veja o Capítulo 27.
Plano de desenvolvimento de pessoal	Descreve como as habilidades e a experiência dos membros da equipe de projeto serão desenvolvidas. Veja o Capítulo 22.

Figura 5.1

Planejamento do projeto.

```
Defina as restrições do projeto
Faça a avaliação inicial dos parâmetros do projeto
Defina os marcos do projeto e os produtos a serem entregues
while projeto não foi concluído ou cancelado loop
        Elabore um cronograma do projeto
        Inicie as atividades de acordo com o cronograma
        Aguarde (por um período)
        Examine o progresso do projeto
        Revise as estimativas de parâmetros do projeto
        Atualize o cronograma do projeto
        Renegocie as restrições do projeto e os produtos a serem entregues
        if (surgirem problemas) then
                Inicie revisão técnica e possível nova revisão
        end if
end loop
```

tamanho e distribuição das funções. Em seguida, você define os marcos de progresso e os produtos a serem entregues. O processo, então, entra em um loop. Deve ser elaborado um cronograma estimado para o projeto e as atividades definidas no cronograma são iniciadas ou liberadas para prosseguimento. Depois de algum tempo (geralmente cerca de duas ou três semanas), você deve examinar o progresso e anotar as discrepâncias em relação ao cronograma. Devido às estimativas iniciais dos parâmetros do projeto serem experimentais, sempre será necessário modificar o plano original.

À medida que as informações se tornam disponíveis, você deve revisar as hipóteses originais e o cronograma do projeto. Se o projeto estiver atrasado, pode ser necessário renegociar as restrições e os produtos a serem entregues com o cliente. Se essa renegociação não for bem-sucedida e o cronograma não puder ser cumprido, uma revisão técnica pode ser considerada. O objetivo dessa revisão é encontrar uma abordagem alternativa que se limite às restrições do projeto e cumpra o cronograma.

Naturalmente, você não deve contar com o fato de que tudo correrá bem. Problemas de alguma natureza quase sempre surgem durante o projeto. Suas hipóteses e cronograma iniciais devem ser pessimistas em vez de otimistas. Deve existir contingência suficiente no plano, de modo que as restrições e os marcos do projeto não precisem ser renegociados o tempo todo no loop de planejamento.

5.2.1 O plano de projeto

O plano de projeto estabelece os recursos disponíveis para o projeto, a estrutura analítica do projeto e um cronograma para realizar o trabalho. Em algumas organizações, o plano de projeto é um único documento que inclui diferentes tipos de planos (Tabela 5.1). Em outros casos, o plano de projeto está relacionado apenas ao processo de desenvolvimento. As referências a outros planos estão incluídas, mas os planos são separados entre si.

A estrutura de plano que descrevo neste livro é adequada ao último tipo de plano. Os detalhes do plano de projeto variam dependendo do tipo do projeto e da organização. No entanto, a maioria dos planos deve incluir as seguintes seções:

1. *Introdução*. Descreve brevemente os objetivos do projeto e estabelece as restrições (por exemplo, orçamento, prazo etc.) que afetam o gerenciamento do projeto.

2. *Organização do projeto*. Descreve o modo como a equipe de desenvolvimento está organizada, as pessoas envolvidas e seus papéis na equipe.
3. *Análise de riscos*. Descreve os possíveis riscos do projeto, a probabilidade da ocorrência desses riscos e as propostas de estratégias de redução de riscos. Explico os princípios de gerenciamento de riscos na Seção 5.4.
4. *Requisitos de recursos de hardware e de software*. Especificam o hardware e o software de apoio necessários para realizar o desenvolvimento. Se o hardware precisar ser comprado, podem ser incluídas as estimativas de preços e o prazo de entrega.
5. *Estrutura analítica*. Estabelece a estrutura analítica do projeto em atividades e identifica os marcos e os produtos a serem entregues com cada atividade. Os marcos e os produtos a serem entregues são explicados na Seção 5.2.2.
6. *Cronograma do projeto*. Apresenta as dependências entre as atividades, o prazo estimado necessário para atingir cada marco e a alocação de pessoas nas atividades.
7. *Mecanismos de monitoração e elaboração de relatórios*. Definem os relatórios de gerenciamento que devem ser produzidos, quando eles devem ser produzidos e os mecanismos de monitoração de projeto usados.

Você deve revisar regularmente o plano durante o projeto. Algumas partes, como o cronograma do projeto, mudarão freqüentemente e outras partes serão mais estáveis. Para simplificar as revisões, você deve organizar o documento em seções separadas que podem ser individualmente substituídas à medida que o plano evolui.

5.2.2 Marcos e produtos a serem entregues

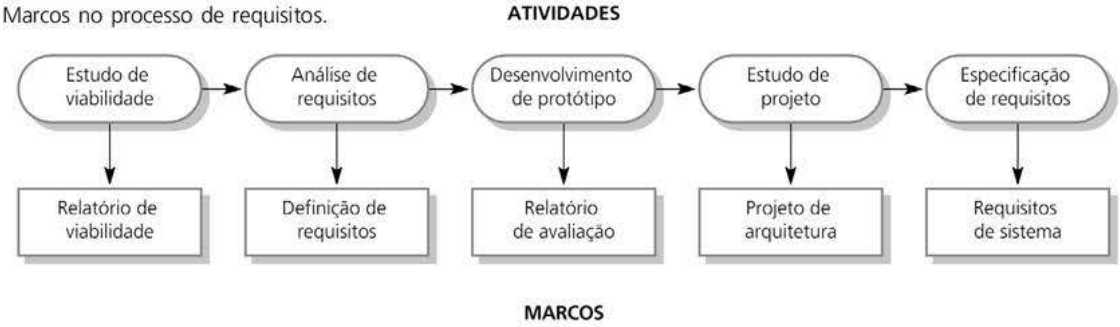
Os gerentes precisam de informações para realizar seu trabalho. Como o software é intangível, essas informações podem ser fornecidas apenas como relatórios e documentos que descrevem o estado do software que é desenvolvido. Sem essas informações, é impossível avaliar quão bem o trabalho está progredindo e as estimativas de custos e o cronograma não podem ser atualizados.

Ao planejar um projeto, você deve estabelecer uma série de *marcos*. Um marco é um ponto final reconhecível de uma atividade do processo de software. A cada marco, deve existir uma saída formal, como um relatório, que possa ser apresentado à gerência. Os relatórios de marcos não precisam ser documentos extensos. Eles podem ser simples relatórios breves sobre o que foi concluído. Os marcos devem representar o fim de um estágio lógico e distinto do projeto. Marcos indefinidos, como 'Codificação 80% concluída', que não podem ser verificados, são inúteis para o gerenciamento do projeto. Você não pode verificar se esse estado foi alcançado, pois a quantidade de código que ainda precisa ser desenvolvida é incerta.

Um produto é um resultado de projeto entregue ao cliente. É geralmente disponibilizado no fim de alguma fase importante do projeto, como especificação ou design. Os produtos são geralmente marcos, mas marcos não precisam ser produtos. Os marcos podem ser resultados internos do projeto usados pelo gerente do projeto para verificar o progresso, embora não sejam entregues ao cliente.

Para estabelecer os marcos, o processo de software deve ser decomposto em atividades básicas com saídas associadas. Por exemplo, a Figura 5.2 mostra as possíveis atividades envolvidas na especificação de requisitos, quando a prototipação é usada para ajudar a validar os requisitos. Os marcos, neste caso, correspondem à finalização das saídas de cada atividade. Os produtos do projeto entregues ao cliente são a definição e a especificação de requisitos.

Figura 5.2 Marcos no processo de requisitos.



5.3 Cronograma do projeto

O desenvolvimento do cronograma do projeto é um dos trabalhos mais difíceis para um gerente de projeto. Os gerentes estimam o tempo e recursos necessários para concluir atividades, organizando-as em uma seqüência coerente. A

menos que o projeto cujo cronograma está sendo desenvolvido seja similar a um projeto anterior, as estimativas anteriores constituem uma base incerta para o desenvolvimento do cronograma do novo projeto. A estimativa de cronograma é mais complicada pelo fato de que projetos diferentes podem usar métodos e linguagens de implementação diferentes.

Se o projeto for tecnicamente avançado, as estimativas iniciais certamente serão otimistas, mesmo quando todas as eventualidades são levadas em conta. Nesse sentido, o desenvolvimento do cronograma de software não é diferente de nenhum outro tipo de projeto avançado de grande porte. Uma nova aeronave, pontes e mesmo novos modelos de automóveis freqüentemente sofrem atrasos devido a problemas não previstos. Os cronogramas, portanto, devem ser continuamente atualizados à medida que mais informações sobre o progresso se tornem disponíveis.

O desenvolvimento do cronograma de projeto (Figura 5.3) envolve a divisão do trabalho total de um projeto em atividades separadas e a avaliação do tempo necessário para completar essas atividades. Em geral, algumas das atividades são realizadas simultaneamente. Você deve coordenar as atividades paralelas e organizar o trabalho de modo que a força de trabalho seja usada de maneira otimizada. É importante evitar uma situação em que todo o projeto fique atrasado devido ao fato de a tarefa crítica não estar concluída.

As atividades de projeto devem durar pelo menos uma semana. Uma subdivisão mais detalhada significa que uma quantidade desproporcional de tempo deve ser despendida na estimativa e na revisão de diagramas. É também útil estabelecer uma quantidade máxima de tempo para qualquer atividade de aproximadamente 8 a 10 semanas. Se ela levar mais tempo que isso, deverá ser subdividida para planejamento e desenvolvimento do cronograma de projeto.

Conforme já sugeri, ao estimar os cronogramas, você não deve considerar que todo o estágio do projeto estará livre de problemas. As pessoas que trabalham em um projeto podem ficar doentes ou podem sair, o hardware pode apresentar defeito e o software ou o hardware de apoio essencial pode ser entregue com atraso. Se o projeto for novo e tecnicamente avançado, certas partes do projeto podem se tornar mais difíceis e tomar mais tempo do que inicialmente previsto.

Além do calendário, você também deve prestar atenção aos recursos necessários para completar cada tarefa. O recurso principal é o esforço humano necessário. Outros recursos podem ser o espaço em disco necessário no servidor, o tempo necessário de um hardware especializado, como um simulador, e o orçamento para viagens necessárias da equipe de projeto. Explico estimativas mais detalhadamente no Capítulo 26.

Uma boa regra prática é fazer a estimativa como se nada fosse dar errado e, então, aumentar a estimativa para cobrir problemas não previstos. Um fator de contingência adicional para cobrir problemas não previstos pode também ser acrescentado à estimativa. Esse fator extra de contingência depende do tipo de projeto, dos parâmetros de projeto (prazo final, padrões etc.) e da qualidade e experiência dos engenheiros de software que trabalham no projeto. Adiciono sempre 30% a minha estimativa original para problemas não previstos e, em seguida, mais 20% para cobrir aspectos sobre os quais não tinha pensado.

Os cronogramas de projeto são geralmente representados como um conjunto de diagramas que apresentam a estrutura analítica do projeto, as dependências de atividades e as alocações de pessoal. Descrevo esses diagramas na seção seguinte. As ferramentas de gerenciamento de projeto, como o Microsoft Project, são geralmente usadas para automatizar a produção de diagramas.

5.3.1 Diagramas de barras e redes de atividades

Os diagramas de barras e as redes de atividades são notações gráficas usadas para ilustrar o cronograma do projeto. Os diagramas de barras mostram quem é responsável por cada atividade e quando as atividades estão programadas para serem iniciadas e terminadas. As redes de atividades mostram as dependências entre as diferentes atividades que constituem um projeto. Os diagramas de barras e os diagramas de atividades podem ser gerados automaticamente baseados em um banco de dados de informações do projeto, usando uma ferramenta de gerenciamento de projetos.

Para ilustrar como os diagramas são usados, criei um conjunto hipotético de atividades, conforme apresentado na Tabela 5.2. Essa tabela mostra as atividades, sua duração e as respectivas interdependências. Ao observar a Tabela 5.2, você pode

Figura 5.3

Processo de desenvolvimento do cronograma do projeto.

Tabela 5.2 Durações e dependências de tarefas

Tarefa	Duração (dias)	Dependências
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

constatar que a atividade T3 é dependente da atividade T1. Isso significa que T1 deve ser terminada antes do início de T3. Por exemplo, T1 pode ser a preparação de um projeto de componente e T3, a implementação desse projeto. Antes do início da implementação, o projeto do componente deve estar concluído.

Em face das dependências e das durações estimadas das atividades, um diagrama de atividades pode ser gerado (Figura 5.4). Ele mostra quais atividades podem ser realizadas simultaneamente e quais devem ser executadas em seqüência devido a uma dependência da atividade anterior. As atividades são representadas por retângulos; os marcos e os produtos a serem entregues são apresentados por retângulos de cantos arredondados. As datas desse diagrama mostram a data de início da atividade. Você deve ler o diagrama da esquerda para a direita e de cima para baixo.

Na ferramenta de gerenciamento de projeto usada para produzir esse diagrama, todas as atividades devem terminar nos marcos. Uma atividade pode ser iniciada quando seu marco precedente (que pode depender de várias atividades) for atingido. Portanto, a terceira coluna da Tabela 5.2 mostra o marco correspondente (por exemplo, M5) alcançado quando as tarefas terminam (veja a Figura 5.4).

Antes de passar de um marco para outro, todos os caminhos que levam para ele precisam estar completos. Por exemplo, quando as atividades T3 e T6 forem terminadas, então a atividade T9, mostrada na Figura 5.4, pode começar.

O tempo mínimo necessário para terminar o projeto pode ser estimado considerando-se o caminho mais longo no gráfico de atividades (o caminho principal). Neste caso, isso corresponde a 11 semanas de tempo decorrido ou 55 dias trabalhados. Na Figura 5.4, o caminho principal é mostrado como uma seqüência de caixas com contorno em negrito. O caminho principal é a seqüência de atividades dependentes que define o tempo necessário para concluir o projeto. O cronograma geral do projeto depende do caminho principal. Qualquer atraso no término de qualquer atividade principal causa atrasos no projeto, porque as atividades seguintes não podem ser iniciadas até que a atividade em atraso tenha sido terminada.

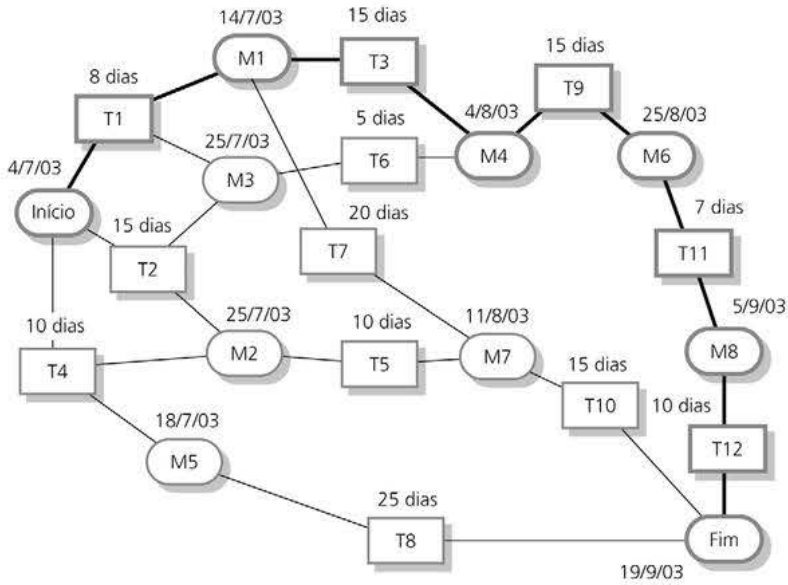
Contudo, os atrasos nas atividades que não estão no caminho principal não causam necessariamente atraso geral no cronograma. Desde que esses atrasos não estendam as atividades de modo que o tempo total para a atividade mais as atividades futuras dependentes não excedam o caminho principal, o cronograma do projeto não será afetado. Por exemplo, se T8 estiver atrasada por duas semanas, ela não irá afetar a data final de término do projeto, pois não está no caminho principal. A maioria das ferramentas de gerenciamento de projetos calcula os atrasos permitidos, conforme mostrado no diagrama de barras do projeto.

Os gerentes também usam os diagramas de atividades ao alocar o trabalho de projeto. Eles podem fornecer esclarecimentos sobre as dependências de atividades que não sejam óbvias. Pode ser possível modificar o projeto de sistema de modo que o caminho principal se torne mais curto. O cronograma de projeto pode se tornar mais curto por causa da redução do tempo gasto que aguarda o término das atividades.

Inevitavelmente, os cronogramas iniciais de projeto serão incorretos. À medida que o projeto avança, as estimativas devem ser comparadas com o tempo real decorrido. Essa comparação pode ser usada como base para revisão de cronograma para as partes posteriores do projeto. Quando os valores reais são conhecidos, o diagrama de atividades deve ser revisado. As atividades posteriores de projeto poderão, depois, ser reorganizadas para reduzir a extensão do caminho principal.

Figura 5.4

Processo de desenvolvimento do cronograma do projeto.



A Figura 5.5 é uma maneira complementar de representação de informações de cronograma do projeto. É um diagrama de barras que mostra um calendário de projeto e as datas de início e fim das atividades. Às vezes é chamado de *diagramas de Gantt*, em homenagem a seu inventor. Ao ser lido da esquerda para a direita, o diagrama de barras mostra claramente quando uma atividade começa e termina.

Algumas das atividades mostradas no diagrama de barras da Figura 5.5 são seguidas por uma barra sombreada cujo comprimento é calculado pela ferramenta de cronograma. Ela demonstra a flexibilidade da data de término das atividades. Se uma atividade não for concluída em tempo, o caminho principal não será afetado até o fim do período indicado pela barra sombreada. As atividades no caminho principal não têm margem de erro e são identificadas por não possuírem barras sombreadas associadas.

Figura 5.5

Diagrama de barras de atividades.

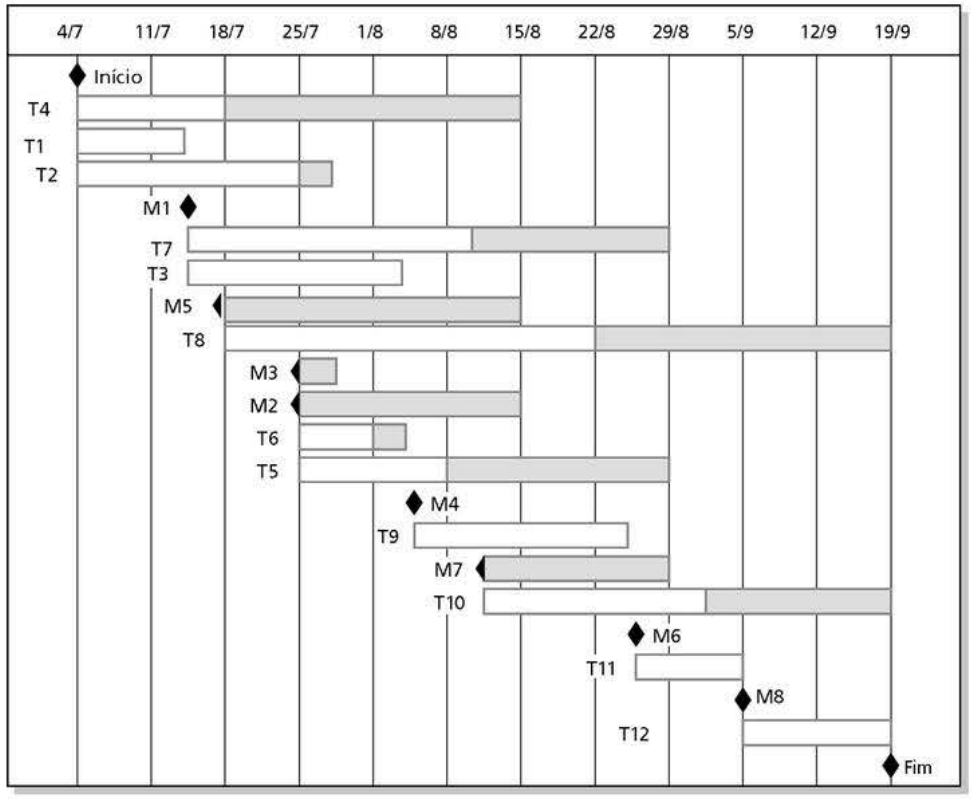
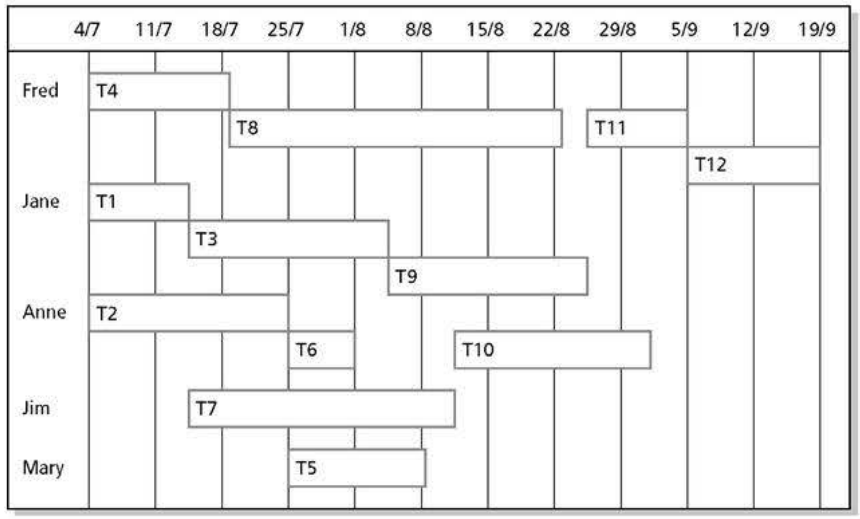


Figura 5.6

Alocação de pessoal *versus* diagrama de tempo.



Além de considerar os cronogramas, os gerentes de projeto também precisam considerar a alocação de recursos e, em particular, a alocação de pessoal às atividades do projeto. Essa alocação pode também ser a entrada para as ferramentas de gerenciamento de projetos e para um diagrama de barras que mostre quando o pessoal é designado ao projeto (Figura 5.6). As pessoas não precisam estar designadas a um projeto durante todo o tempo. Em alguns períodos, elas podem estar em férias, trabalhando em outros projetos, passando por cursos de treinamento ou realizando alguma outra atividade.

Grandes organizações geralmente empregam uma série de especialistas que trabalham em um projeto, se necessário. Na Figura 5.6, observe que Mary e Jim são especialistas que trabalham somente em uma única tarefa no projeto. Isso pode causar problemas no cronograma. Se um projeto atrasa enquanto um especialista está trabalhando nele, isso poderá ter efeito prejudicial sobre os outros projetos. Pode também haver atrasos porque o especialista não está disponível.

5.4 Gerenciamento de riscos

O gerenciamento de riscos está sendo considerado, cada vez mais, como uma das principais atividades dos gerentes de projeto. Ele consiste em prever os riscos que podem afetar o cronograma do projeto ou a qualidade do software que está sendo desenvolvido e tomar providências para evitar esses riscos (Hall, 1998) (Ould, 1999). Os resultados da análise de riscos devem ser documentados no plano de projeto junto com uma análise das conseqüências de uma ocorrência de risco. Um gerenciamento eficiente de riscos torna mais fácil lidar com os problemas e assegurar que eles não conduzirão a um orçamento inaceitável ou atraso no cronograma.

De forma simplificada, você pode pensar em risco como algo que seria preferível não ocorrer. Os riscos podem ameaçar o projeto, o software que está sendo desenvolvido ou a organização. Existem, portanto, três categorias de risco relacionadas:

1. *Riscos de projeto* são aqueles que afetam o cronograma ou os recursos de projeto. Um exemplo pode ser a perda de um projetista experiente.
2. *Riscos de produto* são aqueles que afetam a qualidade ou o desempenho do software que está sendo desenvolvido. Um exemplo pode ser um componente comprado não funcionar conforme esperado.
3. *Riscos de negócio* são aqueles que afetam a organização que desenvolve ou adquire o software. Por exemplo, um concorrente que lança um produto é um risco de negócio.

Naturalmente, esses tipos de risco se sobrepõem. Se um programador experiente deixa um projeto, isso pode ser um risco de projeto, pois a entrega do sistema pode atrasar. Pode também ser um risco de produto, pois um substituto pode não ser tão experiente e, então, pode cometer erros de programação. Finalmente, pode ser um risco de negócio, pois a experiência do programador não está disponível para ser oferecida em negócios futuros.

Os riscos que podem afetar um projeto dependem do projeto e do ambiente organizacional onde o software está sendo desenvolvido. No entanto, muitos riscos são universais. Alguns dos riscos mais comuns são mostrados na Tabela 5.3.

O gerenciamento de riscos é particularmente importante para projetos de software, devido às incertezas inerentes à maioria dos projetos. Elas se originam de requisitos mal definidos, dificuldades na estimativa de prazo e recursos necessários para o desenvolvimento de software, dependência de habilidades individuais e mudanças de requisitos devido às mudanças nas necessidades do cliente. Você precisa prever os riscos, compreender o impacto desses riscos no projeto, no produto e nos

Tabela 5.3 Possíveis riscos de software

Risco	Tipo de risco	Descrição
Rotatividade de pessoal	Projeto	Pessoal experiente deixará o projeto antes do seu término.
Mudança de gerência	Projeto	Haverá uma mudança na gerência da organização com diferentes prioridades.
Indisponibilidade de hardware	Projeto	O hardware essencial ao projeto não será entregue dentro do prazo.
Mudança de requisitos	Projeto e produto	Haverá um número maior de mudanças de requisitos do que o previsto.
Atrasos de especificação	Projeto e produto	As especificações das interfaces essenciais não estarão disponíveis dentro do prazo.
Tamanho subestimado	Projeto e produto	O tamanho do sistema foi subestimado.
Baixo desempenho da ferramenta CASE	Produto	As ferramentas CASE que apóiam o projeto não funcionam conforme previsto.
Mudança de tecnologia	Negócios	A tecnologia sobre a qual o sistema foi construído foi superada por uma nova tecnologia.
Concorrência de produto	Negócios	Um produto concorrente foi lançado no mercado antes da conclusão do sistema.

negócios, e tomar providências para evitar esses riscos. Pode ser necessário elaborar planos de contingência, de modo a poder tomar providências imediatas para recuperação, caso os riscos ocorram.

O processo de gerenciamento de riscos está ilustrado na Figura 5.7. Ele envolve vários estágios:

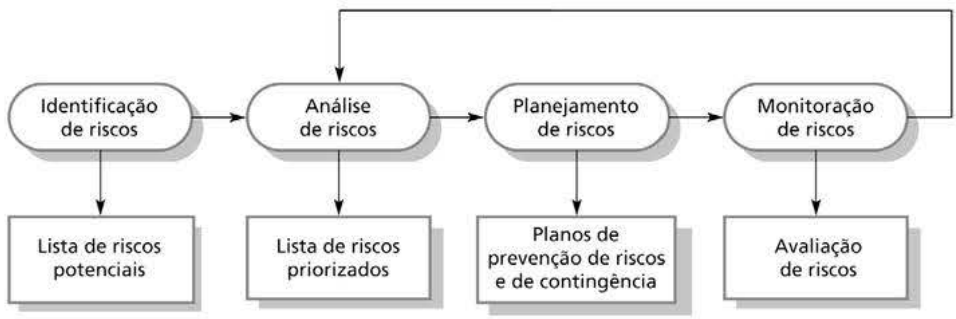
1. *Identificação de riscos*. Os riscos possíveis de projeto, de produto e de negócios são identificados.
2. *Análise de riscos*. A probabilidade e as conseqüências desses riscos são avaliadas.
3. *Planejamento de riscos*. São elaborados planos para cuidar dos riscos seja evitando-os ou minimizando seus efeitos no projeto.
4. *Monitoração de riscos*. Os riscos são constantemente avaliados e os planos para mitigação de riscos são revisados à medida que mais informações se tornam disponíveis.

O processo de gerenciamento de riscos, como todos os outros planejamentos de projeto, é um processo iterativo que prossegue ao longo do projeto. Uma vez elaborado um conjunto inicial de planos, a situação é monitorada. À medida que mais informações sobre os riscos se tornarem disponíveis, os riscos deverão ser analisados novamente e novas prioridades deverão ser estabelecidas. Os planos de prevenção de riscos e de contingência podem ser modificados à medida que novas informações de risco surgirem.

Você deve documentar os resultados do processo de gerenciamento de riscos em um plano de gerenciamento de riscos. Esse plano deve incluir uma explicação dos riscos enfrentados pelo projeto, uma análise desses riscos e os planos necessários para gerenciar esses riscos. Quando apropriado, você também deve incluir no plano os resultados do processo de gerenciamento de riscos, como planos de contingência específicos a serem ativados caso o risco venha a ocorrer.

Figura 5.7

Processo de gerenciamento de riscos.



5.4.1 Identificação de riscos

A identificação de riscos é o primeiro estágio do gerenciamento de riscos. Ela está relacionada com a descoberta dos possíveis riscos do projeto. Em princípio, os riscos não devem ser avaliados ou priorizados neste estágio, embora, na prática, riscos com conseqüências muito pequenas ou com muito pouca probabilidade de ocorrência não sejam normalmente considerados.

A identificação de riscos pode ser realizada como um processo em equipe, usando uma abordagem de brainstorming, ou simplesmente ser baseada na experiência. Para auxiliar o processo, um checklist de diferentes tipos de risco pode ser usado. Existem, pelo menos, seis tipos de risco que podem surgir:

1. *Riscos de tecnologia*. São aqueles que derivam de tecnologias de software ou hardware usadas para desenvolver o sistema.
2. *Riscos de pessoal*. São aqueles associados às pessoas da equipe de desenvolvimento.
3. *Riscos organizacionais*. São aqueles que derivam do ambiente organizacional no qual o software está sendo desenvolvido.
4. *Riscos de ferramentas*. São aqueles que derivam de ferramentas CASE e outros softwares de apoio, usados para desenvolver o sistema.
5. *Riscos de requisitos*. São aqueles que derivam de mudanças de requisitos de cliente e do processo de gerenciamento de mudança de requisitos.
6. *Riscos de estimativas*. São aqueles que derivam de estimativas de gerenciamento das características de sistema e estimativas de recursos necessários para construir o sistema.

A Tabela 5.4 apresenta alguns exemplos dos possíveis riscos em cada uma dessas categorias. Após concluir o processo de identificação de riscos, você deverá ter uma longa lista de riscos que podem ocorrer e quais podem afetar o produto, o processo e os negócios.

5.4.2 Análise de riscos

Durante o processo de análise de riscos, você precisa considerar cada risco identificado e fazer uma avaliação de sua probabilidade e seriedade. Não existe uma maneira fácil de fazer isso — você deve contar com sua própria avaliação e experiência, o que faz com que gerentes de projetos experientes sejam as melhores pessoas para auxiliar no gerenciamento de riscos. Essas estimativas de risco geralmente não precisam ser avaliações numéricas precisas, mas devem basear-se em um número de faixas:

- A probabilidade do risco deve ser avaliada como muito baixa (<10%), baixa (10–25%), média (25–50%), alta (50–75%) ou muito alta (>75%).
- Os efeitos do risco podem ser avaliados como catastróficos, sérios, toleráveis ou insignificantes.

Tabela 5.4 Riscos e tipos de risco.

Tipo de risco	Riscos possíveis
Tecnologia	O banco de dados usado no sistema não pode processar tantas transações por segundo como esperado. Os componentes de software que devem ser reusados contêm defeitos que limitam sua funcionalidade.
Pessoal	É impossível recrutar pessoal com as habilidades necessárias. O pessoal mais qualificado está doente e não disponível nos momentos críticos. O treinamento necessário para o pessoal não está disponível.
Organizacional	A organização é reestruturada, de modo que uma gerência diferente tornou-se responsável pelo projeto. Problemas financeiros da organização forçam reduções no orçamento do projeto.
Ferramentas	O código gerado pelas ferramentas CASE é ineficiente. As ferramentas CASE não podem ser integradas.
Requisitos	Mudanças de requisitos que requerem retrabalho maior de projeto são propostas. Os clientes não compreendem o impacto das mudanças de requisitos.
Estimativas	O prazo necessário para desenvolver o software foi subestimado. A taxa de reparo de defeitos foi subestimada. O tamanho do software foi subestimado.

Você deve, portanto, computar os resultados desse processo de análise usando uma tabela ordenada de acordo com a gravidade do risco. A Tabela 5.5 ilustra isso em relação aos riscos identificados na Figura 5.4. Obviamente, a avaliação de probabilidade e de seriedade é arbitrária neste caso. Na prática, para fazer essa avaliação, você precisa de informações detalhadas sobre o projeto, o processo, a equipe de desenvolvimento e a organização.

Naturalmente, tanto a probabilidade quanto a avaliação dos efeitos de um risco podem mudar à medida que mais informações sobre o risco tornam-se disponíveis e quando os planos de gerenciamento de riscos são implantados. Portanto, essa tabela deve ser atualizada após cada iteração do processo de gerenciamento de riscos.

Após os riscos terem sido analisados e classificados, você deve avaliar quais são os mais significativos. A avaliação deve depender da combinação da probabilidade da ocorrência do risco e de seus efeitos. Em geral, riscos catastróficos devem sempre ser considerados, assim como todos os riscos sérios que tenham probabilidade de ocorrência acima da média.

Boehm (Boehm, 1988) recomenda identificar e monitorar os '10 maiores' riscos, mas penso que esse número é um tanto arbitrário. O número correto de riscos a serem monitorados depende do projeto. Podem ser 5 ou 15. No entanto, o número de riscos selecionados para monitoração deve ser gerenciável. Um número muito grande de riscos exigirá que muitas informações sejam coletadas. Com base nos riscos identificados na Tabela 5.5, é apropriado considerar todos os 8 riscos com conseqüências catastróficas ou sérias.

5.4.3 Planejamento de riscos

O processo de planejamento de riscos considera cada um dos riscos importantes identificados e define estratégias para gerenciá-los. Novamente, não existe um processo simples a ser seguido para definir os planos de gerenciamento de riscos. Ele conta com a avaliação e a experiência do gerente do projeto. A Tabela 5.6 mostra as possíveis estratégias identificadas para os riscos principais da Tabela 5.5.

Essas estratégias dividem-se em três categorias:

1. *Estratégias de prevenção*. Seguir essas estratégias significa que a probabilidade de que o risco ocorrerá será reduzida. Um exemplo de uma estratégia de prevenção de riscos é lidar com componentes defeituosos, como mostrado na Tabela 5.6.
2. *Estratégias de minimização*. Seguir essas estratégias significa que o impacto do risco será reduzido. Um exemplo de estratégia de minimização de riscos é de doença entre o pessoal da equipe como mostrado na Tabela 5.6.

Tabela 5.5 Análise de riscos

Risco	Probabilidade	Efeitos
Problemas financeiros da organização forçam reduções no orçamento do projeto.	Baixa	Catastróficos
É impossível recrutar pessoal com as habilidades necessárias para o projeto.	Alta	Catastróficos
O mais qualificado está doente nos momentos críticos do projeto.	Média	Sérios
Os componentes de software que devem ser reusados contêm defeitos que limitam sua funcionalidade	Média	Sérios
São propostas mudanças de requisitos que requerem maior retrabalho de projeto.	Média	Sérios
A organização é reestruturada, e uma gerência diferente tornou-se responsável pelo projeto.	Alta	Sérios
O banco de dados usado no sistema não pode processar tantas transações por segundo como esperado.	Média	Sérios
O tempo necessário para desenvolver o software foi subestimado.	Alta	Sérios
As ferramentas CASE não podem ser integradas.	Alta	Toleráveis
Os clientes não compreendem o impacto das mudanças de requisitos.	Média	Toleráveis
O treinamento necessário para o pessoal não está disponível.	Média	Toleráveis
A taxa de reparo de defeitos foi subestimada.	Média	Toleráveis
O tamanho do software foi subestimado.	Alta	Toleráveis
O código gerado pelas ferramentas CASE é ineficiente.	Média	Insignificantes

Tabela 5.6 Estratégias de gerenciamento de riscos

Risco	Estratégia
Problemas financeiros da organização	Preparar um documento de instruções para a gerência sênior, que mostre como o projeto está contribuindo de maneira muito importante para as metas da empresa.
Problemas de recrutamento	Alertar o cliente sobre as dificuldades potenciais e a possibilidade de atrasos; investigar a compra de componentes.
Doença do pessoal da equipe	Reorganizar a equipe de maneira que haja mais superposição de trabalho e, portanto, as pessoas compreendam as tarefas uns dos outros.
Componentes com defeito	Substituir os componentes potencialmente defeituosos por componentes comprados e de confiabilidade reconhecida.
Mudanças de requisitos	Derivar informações de rastreabilidade para avaliar o impacto das mudanças de requisitos e maximizar o ocultamento de informações no projeto.
Reestruturação da organização	Preparar um documento de instruções para a gerência sênior que mostre como o projeto está contribuindo de maneira muito importante para as metas da empresa.
Desempenho do banco de dados	Verificar a possibilidade de comprar um banco de dados com desempenho melhor.
Prazo de desenvolvimento subestimado	Verificar a compra de componentes e verificar o uso de um gerador de programa.

3. *Planos de contingência.* Seguir essas estratégias significa que você está preparado para o pior e tem uma estratégia para lidar com o problema. Um exemplo de estratégia de contingência é a estratégia para problemas financeiros da organização na Tabela 5.6.

Note a analogia com as estratégias usadas em sistemas críticos para assegurar confiabilidade, proteção e segurança. Essencialmente, é melhor usar uma estratégia que evite o risco. Se isso não for possível, use uma estratégia que reduza as chances de que o risco tenha sérios efeitos. Finalmente, tenha estratégias que reduzam o impacto geral de ocorrência de um risco no projeto ou no produto.

5.4.4 Monitoração de riscos

A monitoração de riscos envolve a avaliação regular de cada um dos riscos identificados para decidir se esse risco está ou não se tornando mais ou menos provável e se os efeitos do risco mudaram. Naturalmente, isso não pode ser observado diretamente, portanto, é necessário observar outros fatores que forneçam indícios a respeito da probabilidade do risco e seus efeitos. Esses fatores são obviamente dependentes dos tipos de risco. A Tabela 5.7 apresenta alguns exemplos de fatores que podem ser úteis na avaliação desses tipos de risco.

A monitoração de riscos deve ser um processo contínuo e, a cada revisão de progresso feita pela gerência, é necessário considerar e discutir cada um dos principais riscos separadamente.

Tabela 5.7 Fatores de risco

Tipo de risco	Indicadores potenciais
Tecnologia	Entrega de hardware ou software de apoio com atraso, muitos problemas de tecnologia relatados.
Pessoal	Baixo moral do pessoal, relacionamentos precários entre os membros da equipe, disponibilidade de emprego.
Organizacional	Boatos na organização, falta de ação da gerência sênior.
Ferramentas	Relutância dos membros da equipe em usar ferramentas, reclamações sobre ferramentas CASE, demandas por estações de trabalho mais poderosas.
Requisitos	Muitas solicitações de mudança de requisitos, reclamações do cliente.
Estimativas	Falha no cumprimento do cronograma, falha em eliminar defeitos relatados.

PONTOS-CHAVE

- Um bom gerenciamento de projeto de software é essencial para que os projetos de engenharia de software sejam desenvolvidos dentro do cronograma e do orçamento.
- O gerenciamento de software é diferente do gerenciamento em outras áreas da engenharia. O software é intangível. Os projetos podem ser novos ou inovadores, portanto, não existe um conjunto de experiências para guiar seu gerenciamento. Os processos de software não são bem compreendidos.
- Os gerentes de software possuem diversos papéis. As atividades mais significativas são planejamento, elaboração de estimativas e desenvolvimento de cronograma do projeto. O planejamento e a elaboração de estimativas são processos iterativos. Eles continuam ao longo de um projeto. À medida que mais informações se tornam disponíveis, os planos e os cronogramas devem ser revisados.
- Um marco de projeto é um resultado previsível de uma atividade, no qual algum relatório formal de progresso deve ser apresentado à gerência. Os marcos devem ocorrer regularmente ao longo de um projeto de software. Um produto é um marco disponibilizado para o cliente do projeto.
- O desenvolvimento do cronograma do projeto envolve a criação de várias representações gráficas de partes do plano de projeto. Elas incluem diagramas de atividades, que mostram os inter-relacionamentos entre as atividades de projeto, e diagramas de barras, que mostram as durações de atividades.
- Os principais riscos do projeto devem ser identificados e avaliados para definir sua probabilidade e as conseqüências para o projeto. Você deve elaborar planos para evitar, gerenciar ou lidar com os prováveis riscos se ou quando eles ocorrerem. Os riscos devem ser explicitamente discutidos em cada reunião de progresso do projeto.

LEITURAS SUGERIDAS

Waltzing with bears: managing risk on software projects. Uma apresentação muito prática e de fácil leitura a respeito dos riscos e gerenciamento de riscos. (T. DeMarco e T. Lister, 2003, Dorset House.)

Managing software quality and business risk. O Capítulo 3 deste livro é simplesmente a melhor explicação sobre riscos que já vi. O livro trata da questão dos riscos e penso que seja, provavelmente, o melhor livro disponível sobre esse assunto. (M. Ould, 1999, John Wiley & Sons.)

The mythical man month (anniversary edition). Os problemas de gerenciamento de software não mudaram desde a década de 1960 e este é um dos melhores livros sobre o assunto. É um relato interessante e de fácil leitura de um dos primeiros grandes projetos de software, o sistema operacional IBM OS/360. A edição de aniversário (publicada 20 anos após a edição original em 1975) inclui outros artigos clássicos de Brooks. (F. P. Brooks, 1995, Addison-Wesley.)

Software project survival guide. Este é um relato muito pragmático do gerenciamento de software, mas contém recomendações de boas práticas. É fácil de ler e de compreender. (S. McConnell, 1998, Microsoft Press.)

Consulte a Parte 6 para obter outras indicações de leituras sobre gerenciamento.

EXERCÍCIOS

5.1 Explique por que a intangibilidade dos sistemas de software apresenta problemas especiais ao gerenciamento de projetos de software.

5.2 Explique por que os melhores programadores nem sempre se tornam os melhores gerentes de software. Pode ser útil basear sua resposta na lista de atividades de gerenciamento apresentada na Seção 5.1.

5.3 Explique por que o processo de planejamento de software é iterativo e por que um plano deve ser continuamente revisado durante um projeto de software.

5.4 Explique brevemente o propósito de cada uma das seções em um plano de projeto de software.

5.5 Qual é a principal diferença entre um marco e um produto?

5.6 A Tabela 5.8 estabelece uma série de atividades, durações e dependências. Elabore um diagrama de atividades e um diagrama de barras que mostre o cronograma do projeto.

5.7 A Tabela 5.2 apresenta as durações das atividades de projeto de software. Suponha que ocorra um problema sério e imprevisto e, em vez de levar 10 dias, a tarefa T5 leve 40 dias. Revise o diagrama de atividades de acordo com a nova situação, ressaltando o novo caminho principal. Elabore um novo diagrama de barras, que mostre como o projeto pode ser reorganizado.

5.8 Usando os exemplos de problemas de projeto apresentados na literatura, liste as dificuldades de gerenciamento nesses projetos de programação que falharam. (Sugiro que você comece com o livro de Brooks, conforme sugerido em Leituras Sugeridas).

Tabela 5.8 Durações e dependências de tarefas

Tarefa	Duração (dias)	Dependências
T1	10	
T2	15	T1
T3	10	T1, T2
T4	20	
T5	10	
T6	15	T3, T4
T7	20	T3
T8	35	T7
T9	15	T6
T10	5	T5, T9
T11	10	T9
T12	20	T10
T13	35	T3, T4
T14	10	T8, T9
T15	20	T12, T14
T16	10	T15

5.9 Além dos riscos apresentados na Tabela 5.4, identifique seis outros riscos possíveis que podem surgir em projetos de software.

5.10 Os contratos de preço fixo, nos quais o fornecedor oferece um preço fixo para concluir o desenvolvimento de um sistema, podem ser usados para passar o risco do projeto do cliente para o fornecedor. Se algo der errado, o fornecedor deve pagar. Sugira como o uso de tais contratos pode aumentar a probabilidade de ocorrência de riscos de produto.

5.11 Seu gerente solicitou a entrega do software em um cronograma que você sabe que só poderá ser cumprido se sua equipe de projeto trabalhar horas extras sem remuneração. Todos os membros da equipe têm filhos pequenos. Explique se você deve aceitar esse pedido de seu gerente ou se você deve persuadir sua equipe a dar seu tempo para a organização em vez de a suas famílias. Que fatores podem ser significativos para sua decisão?

5.12 Como programador, você recebe uma promoção para gerente de projeto, mas sente que pode prestar uma contribuição mais eficiente na função técnica do que na administrativa. Explique se você deve aceitar a promoção.

PARTE 2 REQUISITOS

Talvez o maior problema que enfrentamos no desenvolvimento de sistemas de software grandes e complexos seja o da engenharia de requisitos. Ela está relacionada com a definição do que o sistema deve fazer, suas propriedades emergentes desejáveis e essenciais e as restrições quanto à operação do sistema e quanto aos processos de desenvolvimento de software. Você pode, portanto, pensar na engenharia de requisitos como o processo de comunicação entre os clientes e os usuários de software e os desenvolvedores de software.

A engenharia de software não é simplesmente um processo técnico. Os requisitos do sistema são influenciados pelas preferências, recusas e preconceitos dos usuários, além das questões políticas e organizacionais. Esses fatores são características humanas fundamentais, e novas tecnologias, como casos de uso, cenários e métodos formais, não nos ajudam muito na resolução desses difíceis problemas.

Os capítulos desta seção dividem-se em duas classes — nos capítulos 6 e 7, apresento as bases da engenharia de requisitos e, nos capítulos 8 a 10, descrevo os modelos e as técnicas usados no processo de engenharia de requisitos. Mais especificamente:

1. O assunto do Capítulo 6 recai sobre os requisitos de software e documentos de requisitos. Explico o que significa um requisito, diferentes tipos de requisitos e como esses requisitos são organizados em um documento de especificação de requisitos. Apresento o segundo estudo de caso usado — um sistema de biblioteca — neste capítulo.
2. No Capítulo 7, enfoco as atividades do processo de engenharia de requisitos. Explico como os estudos de viabilidade devem sempre fazer parte da engenharia de requisitos, técnicas para elicitação e análise de requisitos e validação de requisitos. Como os requisitos inevitavelmente mudam, também abordo o importante tópico de gerenciamento de requisitos.
3. O Capítulo 8 descreve os tipos de modelos de sistema que podem ser desenvolvidos no processo de engenharia de requisitos. Eles fornecem uma descrição mais detalhada para os desenvolvedores de sistema. A ênfase neste capítulo é na modelagem orientada a objetos, mas também incluo uma descrição sobre diagramas de fluxo de dados. Considero esses diagramas intuitivos e úteis, especialmente para fornecer uma visão do início ao fim de como as informações são processadas pelo sistema.
4. A ênfase dos capítulos 9 e 10 é sobre a especificação de sistemas críticos. No Capítulo 9, explico a especificação de propriedades emergentes de confiança. Explico as abordagens dirigidas a riscos e os tópicos específicos de especificação de segurança, de confiabilidade e de proteção. No Capítulo 10, apresento as técnicas formais de especificação. Métodos formais tiveram menos impacto do que o previsto, mas eles estão sendo cada vez mais usados na especificação de sistemas de segurança e de missão crítica. Neste capítulo, explico as abordagens algébricas e baseadas em modelos.

6

Requisitos de software

Objetivos

O objetivo deste capítulo é apresentar os requisitos de sistemas de software e explicar diferentes modos de expressá-los. Ao término da leitura deste capítulo, você:

- compreenderá os conceitos dos requisitos de usuário e de sistema e por que esses requisitos devem ser escritos de maneiras diferentes;
- saberá as diferenças entre os requisitos de software funcionais e não funcionais;
- entenderá como os requisitos podem ser organizados em um documento de requisitos de software.

Conteúdo

6.1 Requisitos funcionais e não funcionais
6.2 Requisitos de usuário
6.3 Requisitos de sistema
6.4 Especificação de interface
6.5 Documento de requisitos de software

Os requisitos de um sistema são descrições dos serviços fornecidos pelo sistema e as suas restrições operacionais. Esses requisitos refletem as necessidades dos clientes de um sistema que ajuda a resolver algum problema, por exemplo, controlar um dispositivo, enviar um pedido ou encontrar informações. O processo de descobrir, analisar, documentar e verificar esses serviços e restrições é chamado de *engenharia de requisitos* (RE — Requirements Engineering). Neste capítulo, concentro-me nos requisitos propriamente ditos e em como descrevê-los. Apresentei o processo de engenharia de requisitos no Capítulo 4, e explico o processo de RE mais detalhadamente no Capítulo 7.

O termo *requisito* não é usado pela indústria de software de maneira consistente. Em alguns casos, um requisito é simplesmente uma declaração abstrata de alto nível de um serviço que o sistema deve fornecer ou uma restrição do sistema. No outro extremo, é uma definição formal e detalhada de uma função do sistema. Davis (Davis, 1993) explica por que essas diferenças existem:

> *Se uma empresa deseja estabelecer um contrato para o desenvolvimento de um grande projeto de software, ela precisa definir suas necessidades de maneira suficientemente abstrata, para que uma solução não seja predefinida. Os requisitos devem ser redigidos de modo que os diversos fornecedores possam apresentar propostas, oferecendo, talvez, diferentes maneiras*

de atender às necessidades organizacionais do cliente. Após a aprovação do contrato, o fornecedor deve redigir uma definição mais detalhada do sistema para o cliente, de modo que o cliente possa compreender e validar o que o software fará. Esses dois documentos podem ser chamados de documentos de requisitos do sistema.

Alguns dos problemas que surgem durante o processo de engenharia de requisitos são resultantes da falta de uma clara separação entre esses diferentes níveis de descrição. Faço essa distinção entre eles usando o termo *requisitos de usuário* para requisitos abstratos de alto nível e *requisitos de sistema* para a descrição detalhada do que o sistema deve fazer. Os requisitos de usuário e de sistema podem ser definidos da seguinte maneira:

1. *Requisitos de usuário* são declarações, em uma linguagem natural com diagramas, de quais serviços são esperados do sistema e as restrições sob as quais ele deve operar.
2. *Requisitos de sistema* definem, detalhadamente, as funções, os serviços e as restrições operacionais do sistema. O documento de requisitos de sistema (às vezes chamado de especificação funcional) deve ser preciso. Ele deve definir exatamente o que será implementado. Pode ser parte do contrato entre o comprador do sistema e os desenvolvedores de software.

Diferentes níveis de especificação de sistema são úteis porque eles comunicam informações sobre o sistema a diferentes tipos de leitores. O Quadro 6.1 ilustra a diferença entre os requisitos de usuário e de sistema. Esse exemplo, baseado em um sistema de biblioteca, mostra como um requisito de usuário pode ser dividido em diversos requisitos de sistema. Você pode notar no Quadro 6.1, que o requisito de usuário é mais abstrato e os requisitos de sistema acrescentam detalhes, explicando os serviços e as funções que devem ser fornecidos pelo sistema a ser desenvolvido.

Você precisa escrever os requisitos em diferentes níveis de detalhes, pois diferentes tipos de leitores usam os requisitos de diferentes maneiras. A Figura 6.1 mostra os tipos de leitores para os requisitos de usuário e de sistema. Os leitores dos requisitos de usuário geralmente não estão preocupados com o modo como o sistema será implementado e podem ser gerentes que não estejam interessados nos recursos detalhados do sistema. Os leitores dos requisitos de sistema necessitam conhecer mais precisamente o que o sistema fará, pois eles estão preocupados com o modo como o sistema apoiará os processos de negócio ou porque estão envolvidos na implementação do sistema.

6.1 Requisitos funcionais e não funcionais

Os requisitos de sistema de software são, freqüentemente, classificados em requisitos funcionais, requisitos não funcionais ou requisitos de domínio:

1. *Requisitos funcionais*. São as declarações de serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais podem também estabelecer explicitamente o que o sistema não deve fazer.
2. *Requisitos não funcionais*. São restrições sobre os serviços ou as funções oferecidos pelo sistema. Eles incluem restrições de timing, restrições sobre o processo de desenvolvimento e padrões. Os requisitos não funcionais aplicam-se,

Quadro 6.1

Requisitos de usuário e de sistema.

Definição de requisitos de usuário

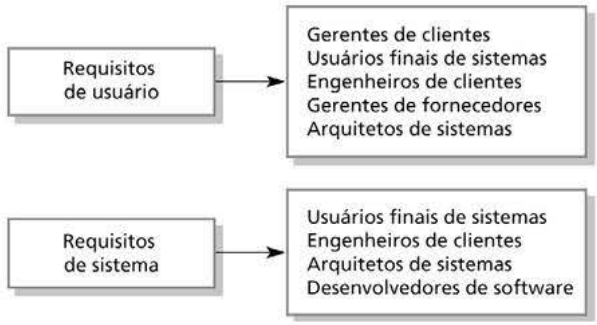
1. LIBSYS deve manter o acompanhamento de todos os dados exigidos pelas agências de licenciamento de direitos autorais no Reino Unido e em outros lugares.

Especificação dos requisitos de sistema

1.1 Ao solicitar um documento ao LIBSYS, deve ser apresentado ao solicitante um formulário que registra os detalhes do usuário e da solicitação feita.
1.2 Os formulários de solicitação do LIBSYS devem ser armazenados no sistema durante cinco anos, a partir da data da solicitação.
1.3 Todos os formulários do LIBSYS devem ser indexados por usuário, nome do material solicitado e fornecedor da solicitação.
1.4 O LIBSYS deve manter um registro de todas as solicitações feitas ao sistema.
1.5 Para materiais aos quais se aplicam os direitos de empréstimo dos autores, os detalhes do empréstimo devem ser enviados mensalmente às agências de licenciamento de direitos autorais que se registraram no LIBSYS.

Figura 6.1

Leitores de diferentes tipos de especificação.



freqüentemente, ao sistema como um todo. Em geral, eles não se aplicam às características ou serviços individuais de sistema.

3. *Requisitos de domínio*. São requisitos provenientes do domínio da aplicação do sistema e que refletem as características e as restrições desse domínio. Podem ser requisitos funcionais ou não funcionais.

Na realidade, a distinção entre os diferentes tipos de requisitos não é tão clara como sugerem essas definições. Um requisito de usuário relacionado à proteção, por exemplo, pode parecer um requisito não funcional. No entanto, quando desenvolvido mais detalhadamente, esse requisito pode gerar outros requisitos claramente funcionais, como a necessidade de incluir recursos de autenticação de usuário no sistema.

6.1.1 Requisitos funcionais

Os requisitos funcionais de um sistema descrevem o que o sistema deve fazer. Esses requisitos dependem do tipo do software que está sendo desenvolvido, dos usuários a que o software se destina e da abordagem geral considerada pela organização ao redigir os requisitos. Quando expressos como requisitos de usuário, eles são geralmente descritos de forma bastante abstrata. No entanto, os requisitos funcionais descrevem a função do sistema detalhadamente, suas entradas e saídas, exceções etc.

Os requisitos funcionais de um sistema de software podem ser expressos de várias formas. Por exemplo, nesta seção, são apresentados os requisitos funcionais de um sistema de biblioteca de uma universidade chamado LIBSYS, usado por estudantes e pela faculdade para solicitar livros e documentos de outras bibliotecas:

1. O usuário deve ser capaz de fazer uma busca em todo o conjunto inicial do banco de dados ou selecionar um subconjunto com base nele.
2. O sistema deve fornecer telas apropriadas para o usuário ler os documentos no repositório de documentos.
3. Para cada pedido, deve ser alocado um único identificador (ORDER_ID), o qual o usuário deve ser capaz de copiar para a área de armazenamento permanente da conta.

Esses requisitos funcionais de usuário definem recursos específicos a serem fornecidos pelo sistema. Eles foram tirados do documento de requisitos de usuário e ilustram que os requisitos funcionais podem ser escritos em níveis diferentes de detalhes (compare os requisitos 1 e 3).

O sistema LIBSYS é uma interface única com uma série de bancos de dados de artigos. Ele permite que os usuários baixem cópias de artigos publicados em revistas, jornais e periódicos científicos. Apresento uma descrição mais detalhada dos requisitos do sistema, nos quais o LIBSYS está baseado, em meu livro com Gerald Kotonya sobre engenharia de requisitos (Kotonya e Sommerville, 1998).

A imprecisão na especificação de requisitos é o motivo de muitos problemas de engenharia de software. É natural que um desenvolvedor de sistema interprete um requisito ambíguo de modo a simplificar sua implementação. Freqüentemente, no entanto, isso não é o que o cliente quer. Novos requisitos precisam ser definidos e mudanças devem ser feitas no sistema. Naturalmente, isso atrasa a entrega do sistema e aumenta os custos.

Considere o segundo requisito do exemplo do sistema de biblioteca, referente a 'telas apropriadas' fornecidas pelo sistema. O sistema de biblioteca pode entregar documentos em uma série de formatos; a intenção desse requisito é que as telas de todos esses formatos estejam disponíveis. No entanto, o requisito está redigido de forma ambígua, sem deixar claro que as telas de cada formato de documento devem ser fornecidas. Um desenvolvedor, sob pressão do cronograma, pode simplesmente fornecer uma tela de texto e argumentar que o requisito foi atendido.

Em princípio, a especificação de requisitos funcionais de um sistema deve ser completa e consistente. *Completeza* significa que todos os serviços exigidos pelo usuário devem ser definidos. *Consistência* significa que os requisitos não devem

ter definições contraditórias. Na prática, em sistemas grandes e complexos, é praticamente impossível atingir a consistência e a completeza de requisitos.

Uma razão para isso é que é fácil cometer erros e omissões quando se redigem especificações para sistemas grandes e complexos. Outra razão é que diferentes *stakeholders* no sistema (veja o Capítulo 7) têm diferentes — e freqüentemente inconsistentes — necessidades. Essas inconsistências podem não ser óbvias quando os requisitos são especificados inicialmente, de modo que requisitos inconsistentes sejam incluídos na especificação. Os problemas podem aparecer apenas depois de uma análise mais profunda ou, às vezes, depois que o desenvolvimento estiver concluído e o sistema tiver sido entregue para o cliente.

6.1.2 Requisitos não funcionais

Os requisitos não funcionais, como o nome sugere, são aqueles não diretamente relacionados às funções específicas fornecidas pelo sistema. Eles podem estar relacionados às propriedades emergentes do sistema, como confiabilidade, tempo de resposta e espaço de armazenamento. Como alternativa, eles podem definir restrições, como a capacidade dos dispositivos de E/S (entrada/saída) e as representações de dados usadas nas interfaces do sistema.

Os requisitos não funcionais estão raramente associados às características individuais do sistema. Pelo contrário, esses requisitos especificam ou restringem as propriedades emergentes de sistema, conforme explicado no Capítulo 2. Portanto, podem especificar desempenho, proteção, disponibilidade e outras propriedades emergentes do sistema. Isso significa que eles geralmente são mais importantes do que os requisitos funcionais individuais. Os usuários do sistema em geral encontram meios de contornar uma função do sistema que não atenda às suas necessidades. Contudo, uma falha no atendimento de um requisito não funcional pode significar que todo o sistema é inútil. Por exemplo, se uma aeronave não atende aos requisitos de confiabilidade, ela não será certificada como segura para operação; se um sistema de controle de tempo real falhar em atender aos requisitos de desempenho, as funções de controle não operarão corretamente.

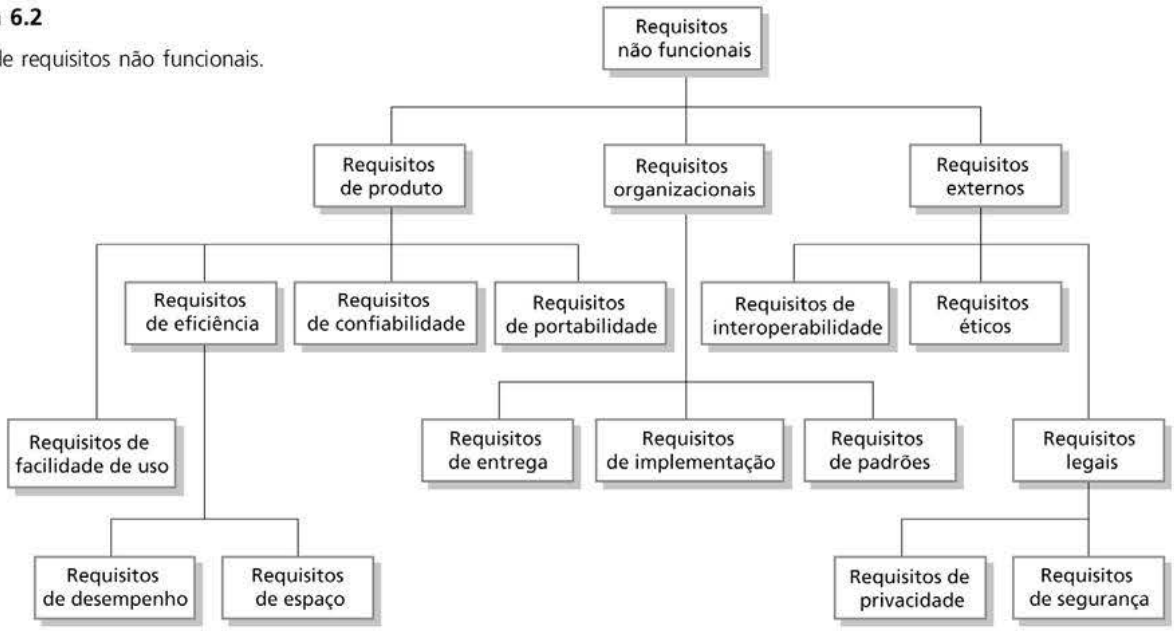
Os requisitos não funcionais não estão relacionados apenas com o sistema de software a ser desenvolvido. Alguns deles podem restringir o processo que deve ser usado para desenvolver o sistema. Exemplos de requisitos de processo incluem uma especificação dos padrões de qualidade que devem ser usados no processo, uma especificação de que o projeto seja produzido com um determinado conjunto de ferramentas CASE e uma descrição do processo que deve ser seguido.

Os requisitos não funcionais surgem devido às necessidades do usuário, às restrições de orçamento, às políticas organizacionais, à necessidade de interoperabilidade com outros sistemas de software ou hardware ou a fatores externos, como regulamentos de segurança ou legislação a respeito da privacidade. A Figura 6.2 apresenta uma classificação de requisitos não funcionais. Você pode observar, no diagrama, que os requisitos não funcionais podem vir de características necessárias do software (requisitos de produto), da organização que desenvolve o software (requisitos organizacionais) ou de fontes externas.

Os tipos de requisitos não funcionais são:

Figura 6.2

Tipos de requisitos não funcionais.



1. *Requisitos de produto.* Estes requisitos especificam o comportamento do produto. Entre os exemplos estão requisitos de desempenho quanto à rapidez com que o sistema deve operar e quanto de memória ele requer, requisitos de confiabilidade que definem a taxa aceitável de falhas, requisitos de portabilidade e requisitos de usabilidade.
2. *Requisitos organizacionais.* Estes requisitos são derivados de políticas e procedimentos da organização do cliente e do desenvolvedor. Alguns exemplos incluem padrões de processo que devem ser usados, requisitos de implementação, como a linguagem de programação ou o método de projeto usado, e requisitos de entrega que especificam quando o produto e a sua documentação devem ser entregues.
3. *Requisitos externos.* Este título amplo abrange todos os requisitos derivados de fatores externos ao sistema e seu processo de desenvolvimento. Podem incluir requisitos de interoperabilidade que definem como o sistema interage com os sistemas em outras organizações, requisitos legais que devem ser seguidos para assegurar que o sistema funciona dentro da lei e requisitos éticos. Os requisitos éticos são aqueles incluídos no sistema para assegurar que ele será aceito por seus usuários e pelo público em geral.

O Quadro 6.2 mostra exemplos de requisitos de produto, organizacionais e externos, tirados do sistema LIBSYS, cujos requisitos de usuário foram explicados na Seção 6.1.1. Um requisito de produto restringe a liberdade dos projetistas do LIBSYS na implementação da interface com o usuário de sistema. Nada é dito sobre a funcionalidade do LIBSYS e isso identifica claramente uma restrição de sistema, e não de função. Esse requisito foi incluído porque simplifica o problema de assegurar que o sistema funcione com diferentes navegadores.

Um requisito organizacional especifica que o sistema deve ser desenvolvido de acordo com um processo padrão da empresa, definido como XYZCo-SP-STAN-95. O requisito externo é derivado da necessidade de o sistema estar em conformidade com a legislação de privacidade. Ele especifica que o pessoal da biblioteca não deve ter acesso a dados como endereços dos usuários do sistema, os quais não são necessários para execução do trabalho.

Um problema comum com os requisitos não funcionais é que eles podem ser difíceis de verificar. Os usuários ou os clientes freqüentemente definem esses requisitos como metas gerais, como usabilidade, capacidade do sistema de se recuperar de falhas ou resposta rápida ao usuário. Essas metas vagas causam problemas para os desenvolvedores de sistema, pois eles deixam o escopo para interpretação e discussão subseqüentes, depois que o sistema é entregue. Como ilustração desse problema, considere o Quadro 6.3. Ele mostra uma meta de sistema relacionada à usabilidade de um sistema de controle de tráfego, uma forma típica de como um usuário pode expressar os requisitos de facilidade de uso. Reescrevi esse requisito para demonstrar como a meta pode ser expressa como um requisito não funcional verificável por teste. Mesmo que seja impossível verificar objetivamente a meta do sistema, você pode projetar testes de sistema para contar os erros cometidos pelos controladores que usam um simulador de sistema.

Sempre que possível, você deve escrever os requisitos não funcionais quantitativamente, de modo que eles possam ser testados objetivamente. A Tabela 6.1 mostra uma série de métricas possíveis de serem usadas para especificar as propriedades

Quadro 6.2

Exemplos de requisitos não funcionais.

Requisito de produto

8.1 A interface de usuário para o LIBSYS deve ser implementada como simples HTML, sem frames ou applets de Java.

Requisito organizacional

9.3.2 O processo de desenvolvimento do sistema e os documentos a serem entregues devem estar em conformidade com o processo e produtos a serem entregues definidos em XYZCo-SP-STAN-95.

Requisito externo

10.6 O sistema não deve revelar quaisquer informações pessoais sobre os usuários do sistema ao pessoal da biblioteca que usa o sistema, com exceção do nome e número de referência da biblioteca.

Quadro 6.3

Metas do sistema e requisitos verificáveis

Meta do sistema

O sistema deve ser fácil de ser usado pelos controladores experientes e ser organizado de modo que os erros dos usuários sejam minimizados.

Requisito não funcional verificável

Os controladores experientes devem ser capazes de usar todas as funções do sistema depois de um treinamento no total de duas horas. Após esse treinamento, o número médio de erros cometidos pelos usuários experientes não deve exceder dois por dia.

Tabela 6.1 Métricas para especificar requisitos não funcionais

Propriedade	Medida
Velocidade	Transações processadas/segundo Tempo de resposta de usuário/evento Tempo de atualização da tela
Tamanho	Kbytes Número de chips de RAM
Facilidade de uso	Tempo de treinamento Número de frames de ajuda
Confiabilidade	Tempo médio de falha Probabilidade de indisponibilidade Taxa de ocorrência de falhas Disponibilidade
Robustez	Tempo para reiniciar após falha Porcentagem de eventos que causam falhas Probabilidade de corrupção de dados por falhas
Portabilidade	Porcentagem de declarações dependentes do sistema-alvo Número de sistemas-alvo

não funcionais do sistema. Você pode medir essas características quando o sistema estiver sendo testado para verificar se ele atendeu ou não aos requisitos não funcionais.

Na prática, no entanto, os clientes de um sistema podem considerar praticamente impossível traduzir suas metas em requisitos quantitativos. Para algumas metas, como facilidade de manutenção, não existem métricas. Em outros casos, mesmo quando uma especificação quantitativa seja possível, os clientes podem não ser capazes de relacionar suas necessidades com essas especificações. Eles não compreendem o que um número que define a confiabilidade necessária, por exemplo, significa em termos de sua experiência diária com sistemas de computador. Além disso, o custo da verificação objetiva dos requisitos não funcionais quantitativos pode ser muito alto e os clientes que pagam pelo sistema podem não considerar que esses custos sejam justificados.

Portanto, os documentos de requisitos freqüentemente incluem declarações de metas combinadas com requisitos. Essas metas podem ser úteis para os desenvolvedores, pois fornecem indicações das prioridades do cliente. Contudo, você deve sempre informar aos clientes que elas estão abertas à má interpretação e não podem ser verificadas objetivamente.

Os requisitos não funcionais freqüentemente entram em conflito e interagem com outros requisitos funcionais e não funcionais. Por exemplo, pode haver um requisito que limite a memória máxima usada por um sistema a 4 Mbytes. As restrições de memória são comuns em sistemas embutidos, no qual o espaço ou o peso é limitado e o número de chips ROM que armazenam o software do sistema deve ser mínimo. Outro requisito pode ser que o sistema seja escrito em Ada, uma linguagem de programação para desenvolvimento de software crítico de tempo real. Entretanto, pode não ser possível compilar um programa em Ada com a funcionalidade necessária em menos que 4 Mbytes. Portanto, precisa haver um compromisso entre esses requisitos: uma linguagem de programação alternativa ou mais memória adicionada ao sistema.

Será útil se você puder diferenciar os requisitos funcionais e não funcionais no documento de requisitos. Na prática, isso é difícil de ser feito. Se os requisitos não funcionais forem definidos separadamente dos funcionais, às vezes será difícil verificar os relacionamentos entre eles. Se eles forem definidos com os requisitos funcionais, você pode ter dificuldade em separar as considerações funcionais e não funcionais e identificar os requisitos relacionados ao sistema como um todo. Contudo, você deve destacar explicitamente os requisitos claramente relacionados com as propriedades emergentes de sistema, como o desempenho e a confiabilidade. Você pode fazer isso colocando-os em uma seção separada do documento de requisitos ou diferenciando-os, de alguma forma, dos demais requisitos de sistema.

Os requisitos não funcionais, como segurança e proteção, são particularmente importantes para sistemas críticos. Portanto, explico os requisitos de confiança mais detalhadamente no Capítulo 9, que aborda a especificação de sistemas críticos.

6.1.3 Requisitos de domínio

Os requisitos de domínio são derivados do domínio de aplicação do sistema, em vez das necessidades específicas dos usuários do sistema. Geralmente incluem uma terminologia específica de domínio ou fazem referência a conceitos do domínio. Podem ser novos requisitos funcionais em si mesmos, podem restringir os requisitos funcionais existentes ou esta-

Quadro 6.4

Requisito de domínio de um sistema de proteção de trens.

A desaceleração do trem deve ser calculada como:

$D_{trem} = D_{controle} + D_{gradiente}$

onde $D_{gradiente}$ é 9,81 ms^2 * gradiente compensado/alfa e onde os valores de 9,81 ms^2/alfa são conhecidos para diferentes tipos de trens.

belecer como cálculos específicos devem ser realizados. Como esses requisitos são especializados, os engenheiros de software freqüentemente encontram dificuldade em compreender como eles estão relacionados a outros requisitos do sistema.

Os requisitos de domínio são importantes porque, com freqüência, refletem os fundamentos do domínio da aplicação. Se esses requisitos não forem satisfeitos, pode ser impossível fazer o sistema funcionar satisfatoriamente. O sistema LIBSYS inclui um conjunto de requisitos de domínio:

1. Deve existir uma interface com o usuário-padrão para todos os bancos de dados e que deverá ser baseada no padrão Z39.50.
2. Devido às restrições de direitos autorais, alguns documentos devem ser excluídos imediatamente na chegada. Dependendo dos requisitos de usuário, esses documentos serão impressos localmente no servidor do sistema para serem encaminhados manualmente para o usuário ou direcionados a uma impressora de rede.

O primeiro requisito é uma restrição de projeto. Ele especifica que a interface com o usuário do banco de dados deve ser implementada de acordo com um padrão específico de biblioteca. Os desenvolvedores, portanto, devem conhecer o padrão antes de iniciar o projeto de interface. O segundo requisito foi introduzido devido às leis de direitos autorais aplicadas ao material usado em bibliotecas. Ele especifica que o sistema deve incluir um recurso automático 'excluir após impressão' para algumas classes de documento. Isso significa que os usuários do sistema de biblioteca não podem ter suas próprias cópias eletrônicas do documento.

Para ilustrar os requisitos de domínio que especificam como um cálculo deve ser realizado, considere o Quadro 6.4, cujos dados são da especificação de requisitos do sistema de proteção de um trem automatizado. Esse sistema pára automaticamente um trem caso ele ultrapasse um sinal vermelho. O requisito define como a desaceleração do trem é calculada pelo sistema. Ele usa a terminologia específica do domínio. Para compreendê-lo, você precisa de algum conhecimento sobre operação de sistemas ferroviários e das características do trem.

O requisito do sistema de trem ilustra um grande problema com os requisitos de domínio. Eles são redigidos na linguagem do domínio da aplicação (equações matemáticas, neste caso), e geralmente os engenheiros de software têm dificuldade em compreendê-los. Os especialistas de domínio podem deixar determinadas informações fora de um requisito, simplesmente por serem muito óbvias para eles. Contudo, pode não ser óbvia para os desenvolvedores do sistema e eles podem, portanto, implementar o requisito de forma equivocada.

6.2 Requisitos de usuário

Os requisitos de usuário de um sistema devem descrever os requisitos funcionais e não funcionais, de modo que eles sejam compreensíveis pelos usuários do sistema que não possuem conhecimento técnico detalhado. Eles devem especificar apenas o comportamento externo do sistema e evitar, sempre que possível, características de projeto do sistema. Conseqüentemente, se você estiver escrevendo requisitos de usuário, você não deve usar jargões de software, notações estruturadas ou formais ou descrever os requisitos por meio da implementação do sistema. Você deve escrever os requisitos de usuário em linguagem simples, com tabelas e formulários simples e diagramas intuitivos.

No entanto, vários problemas podem surgir quando os requisitos são redigidos em um documento com texto em linguagem natural:

1. *Falta de clareza.* Às vezes, é difícil usar a linguagem de maneira precisa e não ambígua sem tornar o documento prolixo e difícil de ler.
2. *Confusão de requisitos.* Requisitos funcionais, requisitos não funcionais, metas de sistema e informações de projeto podem não estar claramente diferenciados.
3. *Fusão de requisitos.* Diversos requisitos diferentes podem ser expressos juntos como um único requisito.

Como ilustração de alguns desses problemas, considere um dos requisitos para a biblioteca apresentado no Quadro 6.5.

Quadro 6.5

Requisito de usuário para um sistema de contabilidade no LIBSYS.

4.5 O LIBSYS deve fornecer um sistema de contabilidade financeira que mantenha registros de todos os pagamentos realizados pelos usuários do sistema. Os gerentes podem configurar esse sistema de modo que os usuários freqüentes possam receber descontos.

Esse requisito inclui informações conceituais e detalhadas. Ele expressa o conceito de que deve haver um sistema de contabilidade como parte inerente de LIBSYS. Contudo, ele também diz que o sistema de contabilidade deve apoiar descontos aos usuários freqüentes de LIBSYS. Esse detalhe seria mais adequado na especificação de requisitos do sistema.

É uma boa prática separar os requisitos de usuário dos requisitos mais detalhados do sistema em um documento de requisitos. De outra forma, os leitores não técnicos dos requisitos de usuário podem ser sobrecarregados pelos detalhes que, na verdade, são relevantes apenas para os técnicos. O Quadro 6.6 ilustra essa confusão. Esse exemplo é um documento real de requisitos para uma ferramenta CASE de edição de modelos de projeto de software. O usuário pode especificar que uma grade deve ser apresentada, de modo que as entidades possam ser posicionadas precisamente em um diagrama.

A primeira sentença mistura três tipos de requisitos:

1. O requisito funcional conceitual define que o sistema de edição deve fornecer uma grade. Ele apresenta uma justificativa lógica para isso.
2. O requisito não funcional fornece informações detalhadas sobre as unidades da grade (centímetros ou polegadas).
3. O requisito não funcional de interface com o usuário define como a grade é ativada e desativada pelo usuário.

O requisito no Quadro 6.6 também apresenta algumas, mas não todas, informações de iniciação. Ele define que a grade inicialmente está desativada. Contudo, não define suas unidades quando é ativada. Ele fornece algumas informações detalhadas — isto é, que o usuário pode alternar entre as duas unidades — mas não o espaçamento entre as linhas da grade.

Os requisitos de usuário que incluem muitas informações restringem a liberdade do desenvolvedor do sistema de apresentar soluções inovadoras para os problemas de usuário e são difíceis de serem compreendidos. O requisito de usuário deve simplesmente enfocar recursos principais a serem fornecidos. Reescrevi o requisito de grade de editor (Quadro 6.7) para enfocar apenas as características essenciais do sistema.

Sempre que possível, você deve tentar associar uma justificativa lógica a cada requisito do usuário. A justificativa deve esclarecer por que o requisito foi incluído, e é particularmente útil quando os requisitos são mudados. Por exemplo, a justificativa lógica no Quadro 6.7 reconhece que uma grade ativa, na qual os objetos posicionados 'saltam' automaticamente para uma linha da grade, pode ser útil. Entretanto, isso foi intencionalmente rejeitado em favor do posicionamento manual. Se uma mudança nesse requisito for proposta mais tarde, ficará claro que o uso de uma grade passiva foi intencional, e não uma decisão de implementação.

Para minimizar os mal-entendidos ao redigir requisitos de usuário, recomendo que você siga algumas diretrizes simples:

1. Invente um formato padrão e assegure-se de que todas as definições de requisitos aderiram a esse formato. A padronização de formato torna as omissões menos prováveis e os requisitos mais fáceis de serem verificados. O formato que

Quadro 6.6

Requisito de usuário para uma grade de editor.

2.6 Recursos de grade Para ajudar no posicionamento de entidades sobre um diagrama, o usuário pode ativar uma grade em centímetros ou em polegadas por meio de uma opção no painel de controle. Inicialmente, a grade não estará ativada. A grade poderá ser ativada ou desativada a qualquer instante durante uma sessão de edição, e poderá ser alternada entre polegadas e centímetros, a qualquer instante. Uma opção de grade será fornecida na visão reduzida, mas o número de linhas de grade mostradas será reduzido para evitar o preenchimento do diagrama

Quadro 6.7

Definição de um recurso de grade de editor.

2.6.1 Recursos de grade

O editor deve fornecer um recurso de grade no qual uma matriz de linhas horizontais e verticais fornece um fundo para a janela do editor. Essa grade deve ser passiva e o alinhamento das entidades é de responsabilidade do usuário.

> *Justificativa lógica:* Uma grade ajuda o usuário a criar um diagrama bem organizado com entidades bem espaçadas. Embora uma grade ativa, na qual as entidades 'saltam' as linhas de grade, possa ser útil, o posicionamento é impreciso. O usuário é a melhor pessoa para decidir onde as entidades devem ser posicionadas.

Especificação: ECLIPSE/WS/Tools/DE/FS Seção 5.6
Fonte: Ray Wilson, escritório de Glasgow

uso mostra o requisito inicial em negrito, inclui uma declaração da justificativa lógica de cada requisito de usuário e uma referência à especificação detalhada de requisitos do sistema. Você pode também incluir informações sobre quem propôs o requisito (a fonte do requisito) de modo que se saiba a quem consultar se o requisito tiver de ser mudado.

2. Use a linguagem de forma consistente. Você deve sempre fazer distinção entre requisitos obrigatórios e desejáveis. *Requisitos obrigatórios* são aqueles a que o sistema deve atender e são escritos com o uso da palavra 'deve'. *Requisitos desejáveis* não são essenciais e são escritos com o uso da palavra 'pode'.
3. Use destaque no texto (negrito, itálico ou cor) para ressaltar as partes principais do requisito.
4. Evitar, sempre que possível, o uso de jargões de informática. Contudo, inevitavelmente aparecerão termos técnicos detalhados nos requisitos de usuário.

Os Robertsons (Robertson e Robertson, 1999), em seu livro que aborda o método VOLERE de engenharia de requisitos, recomendam que os requisitos de usuário sejam inicialmente escritos em cartões, sendo um requisito por cartão. Eles sugerem alguns campos, como as justificativas lógicas dos requisitos, as dependências com outros requisitos, a fonte dos requisitos, material de apoio etc. Isso amplia o formato que usei no Quadro 6.7 e pode ser usado para requisitos de usuário e de sistema.

6.3 Requisitos de sistema

Os requisitos de sistema são versões expandidas dos requisitos de usuário usados pelos engenheiros de software como ponto de partida para o projeto do sistema. Eles adicionam detalhes e explicam como os requisitos de usuário devem ser fornecidos pelo sistema. Podem ser usados como parte do contrato para a implementação do sistema e devem, portanto, ser uma especificação completa e consistente de todo o sistema.

De forma ideal, os requisitos de sistema devem simplesmente descrever o comportamento externo do sistema e suas restrições operacionais. Eles não devem estar relacionados a como o sistema pode ser projetado ou implementado. Entretanto, para especificar completamente um sistema de software complexo no nível de detalhamento necessário, é impossível, na prática, excluir todas as informações de projeto. Existem várias razões para isso:

1. Você pode ter de projetar uma arquitetura inicial do sistema para auxiliar na estruturação da especificação de requisitos. Os requisitos de sistema são organizados de acordo com os diferentes subsistemas que constituem o sistema. Conforme explico nos capítulos 7 e 18, essa definição de arquitetura é essencial caso você deseje reusar componentes de software na implementação do sistema.
2. Na maioria dos casos, os sistemas devem interoperar com outros sistemas existentes. Isso restringe o projeto e essas restrições impõem requisitos ao novo sistema.
3. O uso de uma arquitetura específica para satisfazer os requisitos não funcionais (tal como programação de N-versões para conseguir a confiabilidade, explicada no Capítulo 20) pode ser necessário. Um regulamentador externo que necessita certificar que o sistema é seguro pode especificar que um projeto de arquitetura que já tenha sido certificado seja usado.

A linguagem natural é freqüentemente usada para redigir especificações de requisitos de sistema bem como requisitos de usuário. Contudo, como os requisitos de sistema são mais detalhados que os requisitos de usuário, as especificações em linguagem natural podem ser confusas e difíceis de serem compreendidas:

1. A compreensão da linguagem natural depende do uso das mesmas palavras para os mesmos conceitos pelos leitores e pelos elaboradores da especificação. Isso leva a mal-entendidos devido à ambigüidade da linguagem natural. Jackson (Jackson, 1995) apresenta um excelente exemplo disso ao explicar os avisos apresentados em uma escada rolante. Eles dizem 'Sapatos devem ser usados' e 'Cachorros devem ser carregados'. Deixo para você resolver as interpretações conflitantes dessas frases.
2. Uma especificação de requisitos em linguagem natural é flexível demais. É possível dizer a mesma coisa de maneiras completamente diferentes. Fica por conta do leitor descobrir quando os requisitos são os mesmos e quando são distintos.
3. Não existe uma maneira fácil de padronizar os requisitos em linguagem natural. Pode ser difícil encontrar todos os requisitos relacionados. Para descobrir as conseqüências de uma mudança, pode ser necessário verificar todos os requisitos em vez de apenas um grupo de requisitos relacionados.

Devido a problemas, as especificações de requisitos escritas em linguagem natural são propensas a mal-entendidos. Eles freqüentemente não são descobertos até as fases finais do processo de software e, portanto, a solução deles pode ser muito onerosa.

É essencial escrever os requisitos de usuário em uma linguagem que os não-especialistas possam compreender. Contudo, você pode escrever os requisitos de sistema usando notações mais especializadas (Tabela 6.2). Essas notações incluem linguagem natural estruturada e estilizada, modelos gráficos dos requisitos, como casos de uso até especificações matemáticas formais. Neste capítulo, explico como uma linguagem natural estruturada, suplementada por modelos gráficos simples, pode ser usada para escrever requisitos do sistema. Explico a modelagem gráfica de sistema no Capítulo 8 e a especificação formal do sistema no Capítulo 10.

6.3.1 Especificações em linguagem estruturada

A linguagem natural estruturada é uma forma de escrever os requisitos de sistema, na qual a liberdade do elaborador de requisitos é limitada e todos os requisitos são redigidos de maneira padronizada. A vantagem dessa abordagem é que ela mantém a maior parte da facilidade de expressão e compreensão da linguagem natural, mas assegura que algum grau de uniformidade seja imposto nas especificações. As notações de linguagem estruturada limitam a terminologia que pode ser usada e usa templates para especificar os requisitos de sistema. Elas podem incorporar construções de controle derivadas de linguagens de programação e destaques gráficos para dividir a especificação.

Um antigo projeto que usou a linguagem natural estruturada para especificar os requisitos de sistema é descrito por Heninger (Heninger, 1980). Foram criados formulários com propósitos especiais para descrever a entrada, a saída e as funções do sistema de software de uma aeronave. Os requisitos de sistema foram especificados usando esses formulários.

Para usar uma abordagem baseada em formulários para especificar os requisitos de sistema, é necessário definir um ou mais formulários ou templates-padrão para expressar os requisitos. A especificação pode ser estruturada com base em objetos manipulados pelo sistema, nas funções executadas pelo sistema ou nos eventos processados pelo sistema. Um exemplo de especificação baseada em formulários é apresentado na Tabela 6.3. Esse exemplo é proveniente do sistema de bomba de insulina apresentado no Capítulo 3.

A bomba de insulina baseia seus cálculos da necessidade de insulina do usuário na taxa de mudança dos níveis de açúcar no sangue. Essas taxas de mudança são calculadas por meio das leituras atuais (r2) e anteriores (r1). Você pode baixar a versão completa da especificação da bomba de insulina das páginas Web do livro.

Quando um formulário-padrão é usado para especificar os requisitos funcionais, as seguintes informações devem ser incluídas:

1. Descrição da função ou da entidade que está sendo especificada
2. Descrição de suas entradas e de onde elas são provenientes
3. Descrição de suas saídas e para onde elas prosseguirão
4. Indicações de quais outras entidades são usadas (a parte 'requer')
5. Descrição da ação a ser tomada
6. Se uma abordagem funcional for usada, uma precondição que estabeleça o que deve ser verdadeiro antes da chamada da função, e uma pós-condição que especifique o que é verdadeiro após a chamada da função
7. Descrição dos efeitos colaterais da operação (se existirem).

Tabela 6.2 Notações para especificação de requisitos

Notação	Descrição
Linguagem natural estruturada	Esta abordagem depende da definição de formulários ou templates-padrão para expressar a especificação de requisitos.
Linguagens de descrição de projeto	Esta abordagem usa uma linguagem semelhante à linguagem de programação, porém com mais características abstratas, para especificar os requisitos por meio da definição de um modelo operacional do sistema. Essa abordagem não é amplamente usada hoje em dia, embora possa ser útil para especificações de interfaces.
Notações gráficas	Uma linguagem gráfica, complementada com anotações de texto é usada para definir os requisitos funcionais do sistema. Um antigo exemplo dessa linguagem gráfica é SADT (Ross, 1977) (Schoman e Ross, 1977). Atualmente, as descrições de casos de uso (Jacobsen, et al., 1993) e os diagramas de seqüência são comumente usados (Stevens e Pooley, 1999).
Especificações matemáticas	São notações baseadas em conceitos matemáticos, como máquinas de estados finitos ou conjuntos. Essas especificações não ambíguas reduzem discussões entre cliente e fornecedor. No entanto, a maioria dos clientes não compreende as especificações formais e são relutantes em aceitá-las no momento da contratação.

Tabela 6.3 Especificação de requisitos do sistema com utilização de um formulário-padrão

Bomba de insulina/Software de controle/SRS/3.3.2	
Função	Calcular dose de insulina: nível seguro de açúcar
Descrição	Calcula a dose de insulina a ser liberada quando o nível medido de açúcar atual está na zona segura entre 3 e 7 unidades
Entradas	Leitura atual de açúcar (r2), as duas leituras anteriores (r0 e r1)
Origem	Leitura atual de açúcar do sensor. Outras leituras da memória
Saídas	CompDose — a dose de insulina a ser liberada
Destino	Loop de controle principal
Ação: CompDose será zero se o nível de açúcar estiver estável ou em queda, ou se o nível estiver aumentando, mas a taxa de aumento estiver diminuindo. Se o nível estiver aumentando e a taxa de aumento estiver aumentando, então CompDose será calculado dividindo-se a diferença entre o nível atual de açúcar e o nível anterior por 4, e arredondando o resultado. Se o resultado do arredondamento for zero, então CompDose será definido como a dose mínima que pode ser liberada.	
Requer	Duas leituras anteriores de modo que a taxa de mudança do nível de açúcar possa ser calculada.
Precondição	O reservatório de insulina conter, pelo menos, o máximo de dose única permitida de insulina.
Pós-condição	r0 é substituído por r1, portanto r1 é substituído por r2
Efeitos colaterais	Nenhum

O uso de especificações formatadas elimina alguns dos problemas da especificação em linguagem natural. A variação na especificação é reduzida e os requisitos são organizados de maneira mais eficaz. Contudo, é difícil redigir os requisitos de maneira não ambígua, particularmente quando são necessárias computações complexas. Você pode observar isso na descrição apresentada na Tabela 6.3, na qual não fica claro o que acontecerá se a precondição não for satisfeita.

Para resolver esse problema, você pode adicionar informações extras para os requisitos em linguagem natural, usando tabelas ou modelos gráficos do sistema. Eles podem mostrar como as computações são processadas, como o estado do sistema muda, como os usuários interagem com o sistema e como as seqüências das ações são realizadas.

As tabelas são particularmente úteis quando há uma série de possíveis situações alternativas e você necessita descrever as ações a serem tomadas a respeito de cada uma delas. A Tabela 6.4 é uma descrição revisada do cálculo da dose de insulina.

Os modelos gráficos são mais úteis quando você necessita mostrar como o estado muda (veja o Capítulo 8) ou quando você necessita descrever uma seqüência de ações. A Figura 6.3 ilustra a seqüência das ações quando um usuário deseja retirar dinheiro de um caixa eletrônico.

Você deve ler um diagrama de seqüência de cima para baixo para visualizar a ordem de ocorrência das ações. Na Figura 6.3 existem três subseqüências básicas:

1. *Validar cartão*. O cartão do usuário é validado com a verificação do número do cartão e do número de identificação pessoal.
2. *Tratar solicitação*. A solicitação do usuário é tratada pelo sistema. Para uma retirada, o banco de dados deve ser consultado para verificar o saldo do usuário e debitar a quantidade retirada. Observe que a exceção ocorre se o requisitante não tiver dinheiro suficiente em sua conta.
3. *Completar transação*. O cartão do usuário é devolvido e, quando retirado, o dinheiro e o recibo são entregues.

Você verá o diagrama de seqüência novamente no Capítulo 8, que aborda os modelos de sistema, e no Capítulo 14, que aborda o projeto orientado a objetos.

6.4 Especificação de interface

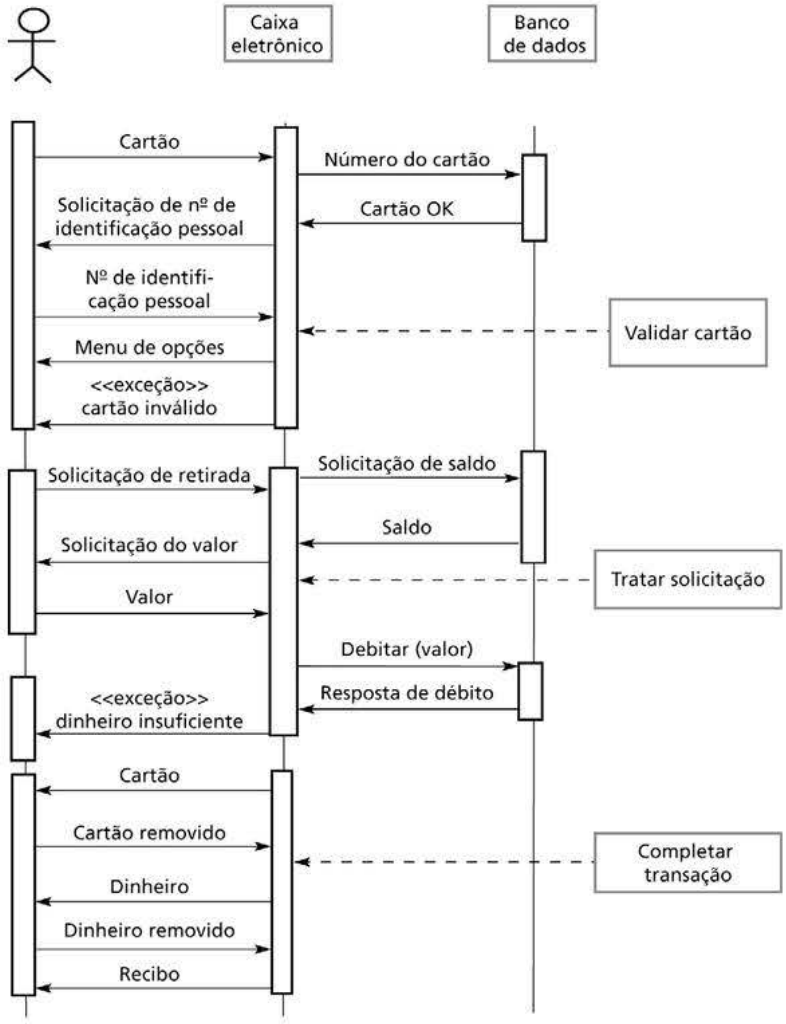
Quase todos os sistemas de software devem operar com sistemas existentes já implementados e instalados em um ambiente. Se o novo sistema deve funcionar junto com os sistemas existentes, as interfaces dos sistemas existentes devem ser especificadas com precisão. Essas especificações devem ser definidas no início do processo e incluídas (talvez como um apêndice) no documento de requisitos.

Tabela 6.4 Especificação tabular de cálculo

Condição	Ação
Nível de açúcar em queda (r2 < r1)	CompDose = 0
Nível de açúcar estável (r2 = r1)	CompDose = 0
Nível de açúcar aumentando e taxa de aumento diminuindo ((r2 – r1) < (r1 – r0))	CompDose = 0
Nível de açúcar aumentando e taxa de aumento estável ou aumentando ((r2 – r1) > (r1 – r0))	CompDose = arredondar ((r2 – r1)/4) Se o resultado do arredondamento for = 0 então CompDose = DoseMínima

Figura 6.3

Diagrama de seqüência de retirada no caixa eletrônico.



Existem três tipos de interface que podem ser definidos:

1. *Interfaces de procedimentos* nas quais programas ou subsistemas existentes oferecem uma série de serviços acessados pela chamada de procedimentos de interface. Essas interfaces são algumas vezes chamadas de Application Programming Interfaces (APIs).
2. *Estruturas de dados* que são passadas de um subsistema para outro. Modelos gráficos de dados (descritos no Capítulo 8) são as melhores notações para esse tipo de descrição. Se forem necessárias, descrições de programa em Java ou C++ podem ser geradas automaticamente com base nessas descrições.
3. *Representações de dados* (como a ordenação de bits) estabelecidas para um subsistema preexistente. Essas interfaces são mais comuns em sistemas incorporados de tempo real. Algumas linguagens de programação, como Ada (embora

Figura 6.4

Descrição PDL em Java da interface do servidor de impressão.

```
interface PrintServer {

// define um servidor abstrato de impressora
// requer: interface Printer, interface PrintDoc
// fornece: inicia, imprime, apresentaPrintQueue, cancelaPrintJob, trocaPrinter

   void inicia ( Printer p ) ;
   void imprime ( Printer p, PrintDoc d ) ;
   void apresentaPrintQueue ( Printer p ) ;
   void cancelaPrintJob (Printer p, PrintDoc d) ;
   void transferePrinter (Printer p1, Printer p2, PrintDoc d) ;
} //PrintServer
```

não Java), apóiam esse nível de especificação. No entanto, a melhor forma de descrever essas interfaces é, provavelmente, o uso de um diagrama de estrutura com anotações que expliquem a função de cada grupo de bits.

As notações formais, explicadas no Capítulo 10, permitem que as interfaces sejam definidas de maneira não ambígua, mas devido à sua natureza especializada não podem ser compreendidas sem um treinamento especial. Elas são raramente usadas na prática para especificação de interfaces embora, em minha opinião, adaptem-se idealmente a esse propósito. Uma linguagem de programação, como Java, pode ser usada para descrever a sintaxe da interface. No entanto, essa descrição deve ser suplementada por descrições adicionais que explicam a semântica de cada uma das operações definidas.

A Figura 6.4 é um exemplo de uma definição de interface de procedimentos em Java. Nesse caso, a interface é aquela oferecida por um servidor de impressão que gerencia uma fila de solicitações para imprimir arquivos em impressoras diferentes. Os usuários podem examinar a fila associada à impressora e remover suas tarefas de impressão dessa fila. Eles podem também transferir as tarefas de uma impressora para outra. A especificação da Figura 6.4 é um modelo abstrato de servidor de impressão que não revela nenhum detalhe da interface. A funcionalidade das operações de interface pode ser definida por meio de linguagem natural estruturada ou descrição tabular.

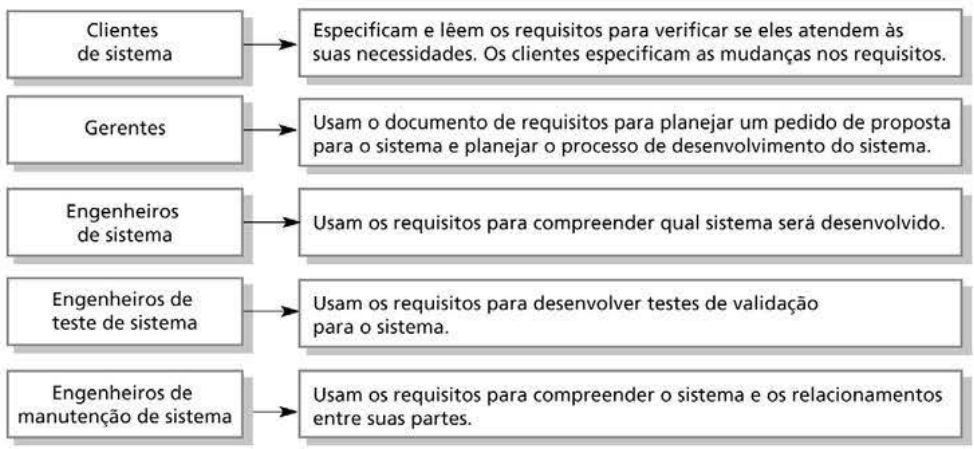
6.5 Documento de requisitos de software

O documento de requisitos de software (algumas vezes chamado de especificação de requisitos de software ou SRS — Software Requirements Specification) é a declaração oficial do que os desenvolvedores de sistema devem implementar. Deve incluir os requisitos de usuário de um sistema e uma especificação detalhada dos requisitos de sistema. Em alguns casos, os requisitos de usuário e de sistema podem estar integrados em uma única descrição. Em outros casos, os requisitos de usuário estão definidos em uma introdução à especificação dos requisitos de sistema. Se houver um grande número de requisitos, os requisitos detalhados de sistema podem ser apresentados em um documento separado.

O documento de requisitos possui um conjunto diversificado de usuários, desde a gerência sênior da organização, que paga pelo sistema, até os engenheiros responsáveis pelo desenvolvimento do software. A Figura 6.5, extraída de meu livro com Gerald Kotonya sobre engenharia de requisitos (Kotonya e Sommerville, 1998), ilustra os possíveis usuários do documento e como eles o utilizam.

Figura 6.5

Usuários de um documento de requisitos.



A diversidade de possíveis usuários significa que o documento de requisitos precisa ser um compromisso entre a comunicação dos requisitos para os clientes, a definição dos requisitos em detalhes precisos para os desenvolvedores e testadores e a inclusão de informações sobre uma possível evolução do sistema. As informações sobre mudanças previstas podem auxiliar os projetistas a evitar decisões restritivas de projeto e auxiliar os engenheiros de manutenção de sistema que devem adaptar o sistema a novos requisitos.

O nível de detalhamento a ser incluído em um documento de requisitos depende do tipo de sistema que está sendo desenvolvido e do processo de desenvolvimento usado. Quando o sistema for desenvolvido por um fornecedor externo, as especificações de sistema crítico devem ser precisas e muito detalhadas. Quando houver maior flexibilidade nos requisitos e quando um processo de desenvolvimento interno e iterativo for usado, o documento de requisitos pode ser muito menos detalhado e qualquer ambigüidade será resolvida durante o desenvolvimento do sistema.

Uma série de grandes organizações, como o Departamento de Defesa dos EUA e o IEEE, definiram padrões para documentos de requisitos. Davis (Davis, 1993) explica alguns desses padrões e compara seus conteúdos. O padrão mais amplamente conhecido é IEEE/ANSI 830-1998 (IEEE, 1998). O padrão do IEEE sugere a seguinte estrutura para os documentos de requisitos:

1. **Introdução**
   - 1.1 Propósito do documento de requisitos
   - 1.2 Escopo do produto
   - 1.3 Definições, acrônimos e abreviaturas
   - 1.4 Referências
   - 1.5 Visão geral do restante do documento
2. **Descrição geral**
   - 2.1 Perspectiva do produto
   - 2.2 Funções do produto
   - 2.3 Características dos usuários
   - 2.4 Restrições gerais
   - 2.5 Suposições e dependências
3. **Requisitos específicos** abrangem requisitos funcionais, não funcionais e de interface. Esta é, obviamente, a parte mais substancial no documento, mas, devido à grande variação na prática organizacional, não é apropriado definir uma estrutura-padrão para esta seção. Os requisitos podem documentar interfaces externas, descrever a funcionalidade e o desempenho do sistema, especificar requisitos lógicos de banco de dados, restrições de projeto, propriedades emergentes do sistema e características de qualidade.
4. **Apêndices**
5. **Índice**

Embora o padrão do IEEE não seja ideal, ele contém uma grande quantidade de boas recomendações de como redigir requisitos e evitar problemas. Ele é muito geral para funcionar como padrão de uma organização. É um framework geral que pode ser configurada e adaptada para definir um padrão dirigido às necessidades de determinada organização. A Tabela 6.5 ilustra uma possível organização para um documento de requisitos com base no padrão do IEEE. No entanto, estendi o padrão para incluir informações sobre a evolução prevista do sistema. Isso foi inicialmente proposto por Heninger (Heninger, 1980) e, conforme expliquei anteriormente, auxilia o pessoal de manutenção do sistema e permite que os projetistas incluam as bases para futuras características do sistema.

Naturalmente, as informações incluídas em um documento de requisitos devem depender do tipo de software que está sendo desenvolvido e da abordagem usada para o desenvolvimento. Se uma abordagem evolucionária for adotada para um produto de software, por exemplo, o documento de requisitos deixará de fora vários capítulos detalhados sugeridos anteriormente. O foco será a definição dos requisitos de usuário e dos requisitos não funcionais de alto nível do sistema. Nesse caso, os projetistas e os programadores usam suas avaliações para decidir como atender aos requisitos de usuários delineados para o sistema.

Por outro lado, quando o software é parte de um grande projeto de engenharia que inclui a interação de sistemas de hardware e de software, freqüentemente é indispensável definir os requisitos em um nível refinado de detalhes. Isso significa que os documentos de requisitos serão, provavelmente, muito extensos e devem incluir a maior parte dos, se não todos, capítulos mostrados na Tabela 6.5. Para documentos extensos, é particularmente importante incluir uma tabela abrangente de conteúdo e índice do documento, de modo que os leitores possam encontrar as informações de que precisam.

Os documentos de requisitos são indispensáveis quando um fornecedor externo está desenvolvendo o sistema de software. No entanto, os métodos ágeis de desenvolvimento argumentam que os requisitos mudam tão rapidamente que um documento de

Tabela 6.5 Estrutura de um documento de requisitos

Capítulo	Descrição
Prefácio	Deve definir o público-alvo do documento e descrever seu histórico de versões, incluindo uma justificativa lógica para a criação da nova versão e um resumo das mudanças feitas em cada versão.
Introdução	Deve descrever a necessidade do sistema. Deve descrever brevemente suas funções e explicar como o sistema irá funcionar com outros sistemas. Deve descrever como o sistema atende aos objetivos gerais de negócios e estratégicos da organização que encomendou o software.
Glossário	Deve definir os termos técnicos usados no documento. Você não deve fazer suposições sobre a experiência ou as habilidades do leitor.
Definição de requisitos de usuário	Os serviços fornecidos ao usuário e os requisitos não funcionais do sistema devem ser descritos nesta seção. Essa descrição pode usar linguagem natural, diagramas e outras notações compreensíveis pelos clientes. Padrões de produto e de processo a serem seguidos devem ser especificados.
Arquitetura de sistema	Este capítulo deve apresentar uma visão geral de alto nível da arquitetura prevista do sistema, mostrando a distribuição das funções nos módulos do sistema. Os componentes de arquitetura reusados devem ser destacados.
Especificação de requisitos de sistema	Deve descrever os requisitos funcionais e não funcionais mais detalhadamente. Caso necessário, mais detalhes podem também ser adicionados aos requisitos não funcionais; por exemplo, interfaces com outros sistemas devem ser definidas.
Modelos de sistema	Deve estabelecer um ou mais modelos de sistema, mostrando os relacionamentos entre os componentes e o sistema e seu ambiente. Podem ser modelos de objetos, modelos de fluxos de dados e modelos semânticos de dados.
Evolução de sistema	Deve descrever as hipóteses fundamentais sobre as quais o sistema está baseado, além de mudanças previstas devido à evolução do hardware, mudança das necessidades do usuário etc.
Apêndices	Deve fornecer informações detalhadas e específicas relacionadas à aplicação que está sendo desenvolvida. Exemplos de apêndices que podem ser incluídos são descrições de hardware e de banco de dados. Os requisitos de hardware definem as configurações mínima e ideal para o sistema. Os requisitos de banco de dados definem a organização lógica dos dados usados pelo sistema e os relacionamentos entre os dados.
Índice	Podem ser incluídos diversos índices para o documento. Assim como um índice alfabético normal, pode haver um índice dos diagramas, índice das funções etc.

requisitos fica desatualizado tão logo seja redigido, por isso o esforço é desperdiçado. Em vez de um documento formal, as abordagens como *extreme programming* (Beck, 1999) propõem que os requisitos de usuário sejam coletados de maneira incremental e escritos em cartões. O usuário, então, prioriza os requisitos a serem implementados no próximo incremento do sistema.

Para sistemas de negócios, nos quais os requisitos são instáveis, penso que esta seja uma boa abordagem. Contudo, argumentaria que é útil ainda escrever um documento breve de apoio que defina os requisitos de negócio e de confiança do sistema. É fácil esquecermos os requisitos que se aplicam ao sistema como um todo quando enfocamos os requisitos funcionais para a próxima versão do sistema.

PONTOS-CHAVE

- Os requisitos de um sistema de software definem o que o sistema deve fazer e as restrições sobre suas operações e sua implementação.
- Os requisitos funcionais são declarações dos serviços que o sistema deve fornecer ou são descrições de como algumas computações devem ser realizadas. Os requisitos de domínio são requisitos funcionais derivados das características do domínio da aplicação.
- Os requisitos não funcionais restringem o sistema que está sendo desenvolvido e o processo de desenvolvimento que deve ser usado. Eles podem ser requisitos de produto, requisitos organizacionais ou requisitos externos. Estão, freqüentemente, relacionados às propriedades emergentes do sistema e, portanto, aplicam-se ao sistema como um todo.
- Os requisitos de usuário destinam-se às pessoas envolvidas no uso e na aquisição do sistema. Eles devem ser redigidos com o uso de linguagem natural, tabelas e diagramas de fácil compreensão.

- Os requisitos de sistema destinam-se a comunicar, de maneira precisa, as funções que o sistema deve fornecer. Para reduzir a ambigüidade, podem ser escritos em linguagem natural, de maneira estruturada, e complementados com tabelas e modelos de sistemas.
- O documento de requisitos de software é a declaração aprovada dos requisitos de sistema. Ele deve ser organizado de tal modo que os clientes de sistema e os projetistas de software possam usá-lo.
- O padrão do IEEE para documentos de requisitos é um ponto de partida útil para padrões de especificação de requisitos mais específicos.

LEITURAS SUGERIDAS

Software requirements, 2nd ed. Este livro, dirigido a elaboradores e usuários de requisitos, explica boas práticas de engenharia de requisitos. (K. M. Weigers, 2003, Microsoft Press.)

Mastering the requirements process. Um livro bem escrito, de fácil leitura, baseado em um método específico (VOLERE), mas que também inclui muitas boas recomendações gerais sobre engenharia de requisitos. (S. Robertson e J. Robertson, 1999, Addison-Wesley.)

Requirements engineering: processes and techniques. Este livro aborda todos os aspectos do processo de engenharia de requisitos e explica técnicas de especificação de requisitos. (G. Kotonya e I. Sommerville, 1999, John Wiley & Sons.)

Software requirements engineering. Esta coleção de artigos sobre engenharia de requisitos inclui vários artigos relevantes, tal como "Recommended practice for software requirements specification", uma explicação do padrão do IEEE para documentos de requisitos. (R. H. Thayer e M. Dorfman (eds.), 1997, IEEE Computer Society Press.)

EXERCÍCIOS

6.1 Identifique e descreva brevemente quatro tipos de requisitos que podem ser definidos para um sistema baseado em computador.

6.2 Explique os problemas do uso de linguagem natural para definição de requisitos de usuário e de sistema e mostre, usando pequenos exemplos, como a estruturação de linguagem natural em formulários pode ajudar a evitar algumas dessas dificuldades.

6.3 Descubra ambigüidades e omissões na seguinte declaração de requisitos de parte de um sistema de emissão de passagens.

Um sistema automático de emissão de passagens vende passagens de trem. Os usuários selecionam seu destino e inserem um cartão de crédito e um número de identificação pessoal. A passagem de trem é emitida e é debitada na conta do cartão de crédito. Quando o usuário pressiona o botão iniciar, uma tela de menu com possíveis destinos é ativada, junto com uma mensagem que solicita ao usuário selecionar um destino. Quando o destino é selecionado, solicita-se ao usuário a inserção de seu cartão de crédito. A validade do cartão é verificada e é solicitado ao usuário que insira um identificador pessoal. Quando a transação do cartão é validada, a passagem é emitida.

6.4 Reescreva a descrição anterior usando a abordagem estruturada descrita neste capítulo. Resolva as ambigüidades identificadas de maneira apropriada.

6.5 Desenhe um diagrama de seqüência que mostre as ações realizadas no sistema de emissão de passagens. Podem-se supor quaisquer hipóteses razoáveis sobre o sistema. Preste atenção especial à especificação dos erros de usuário.

6.6 Usando a técnica sugerida neste capítulo, na qual a linguagem natural é apresentada de maneira padronizada, escreva requisitos plausíveis de usuário para as seguintes funções:

- A função de liberar dinheiro em um caixa eletrônico de banco
- A função de verificação e correção de ortografia em um processador de texto
- Um sistema de bomba de gasolina de auto-atendimento que inclui um leitor de cartão de crédito. O cliente passa o cartão pelo leitor e, então, especifica a quantidade de combustível solicitada. O combustível é fornecido e o débito vai para a conta do cliente.

6.7 Descreva quatro tipos de requisitos não funcionais que podem ser definidos para um sistema. Forneça exemplos de cada um dos tipos de requisitos.

6.8 Escreva um conjunto de requisitos não funcionais para o sistema de emissão de passagens, estabelecendo a confiabilidade esperada e o tempo de resposta.

6.9 Sugira como um engenheiro responsável por definir os requisitos de um sistema pode manter o acompanhamento dos relacionamentos entre os requisitos funcionais e não funcionais.

6.10 Você conseguiu um emprego com um usuário de software que contratou seu empregador anterior para desenvolver um sistema para eles. Descobre depois que a interpretação dos requisitos da empresa é diferente da interpretação dada por seu empregador anterior. Explique o que você deve fazer em tal situação. Você sabe que os custos para seu atual empregador irão aumentar caso as ambigüidades não sejam resolvidas. Também tem a responsabilidade da confidencialidade em relação a seu empregador anterior.

7

Processos de engenharia de requisitos

Objetivos

O objetivo deste capítulo é explicar as atividades envolvidas no processo de engenharia de requisitos. Após ler este capítulo, você:

- compreenderá as principais atividades da engenharia de requisitos e seus relacionamentos;
- terá sido apresentado às diversas técnicas de elicitação e análise de requisitos;
- compreenderá a importância da validação de requisitos e como as revisões de requisitos são usadas nesse processo;
- compreenderá por que o gerenciamento de requisitos é necessário e como ele apóia as outras atividades de engenharia de requisitos.

Conteúdo

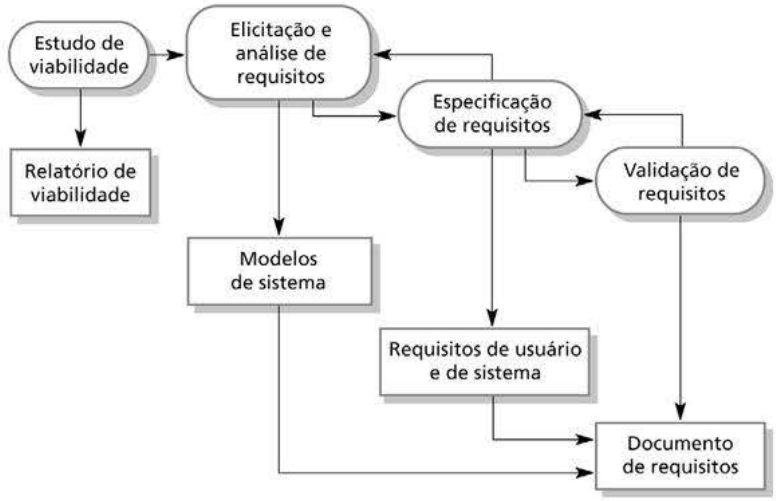
7.1 Estudos de viabilidade
7.2 Elicitação e análise de requisitos
7.3 Validação de requisitos
7.4 Gerenciamento de requisitos

O objetivo do processo de engenharia de requisitos é criar e manter um documento de requisitos de sistema. O processo geral inclui quatro subprocessos de alto nível de engenharia de requisitos. Eles estão relacionados à avaliação de se o sistema é útil para a empresa (estudo de viabilidade), obtenção de requisitos (elicitação e análise), conversão desses requisitos em alguma forma-padrão (especificação) e verificação de se os requisitos realmente definem o sistema que o cliente deseja (validação). A Figura 7.1 ilustra o relacionamento entre essas atividades. Ela mostra também os documentos produzidos em cada estágio do processo de engenharia de requisitos. A especificação e a documentação foram abordadas no Capítulo 6; este capítulo concentra-se nas outras atividades da engenharia de requisitos.

As atividades mostradas na Figura 7.1 estão relacionadas à obtenção, documentação e verificação de requisitos. Na prática, no entanto, todos os requisitos de sistema mudam. As pessoas envolvidas desenvolvem uma compreensão maior do que desejam que o software faça, a organização que está comprando o sistema muda, são feitas modificações no hardware, no software e no ambiente organizacional do sistema. O processo de gerenciamento dessas mudanças de requisitos é denominado gerenciamento de requisitos e é abordado na seção final deste capítulo.

Figura 7.1

Processo de engenharia de requisitos.

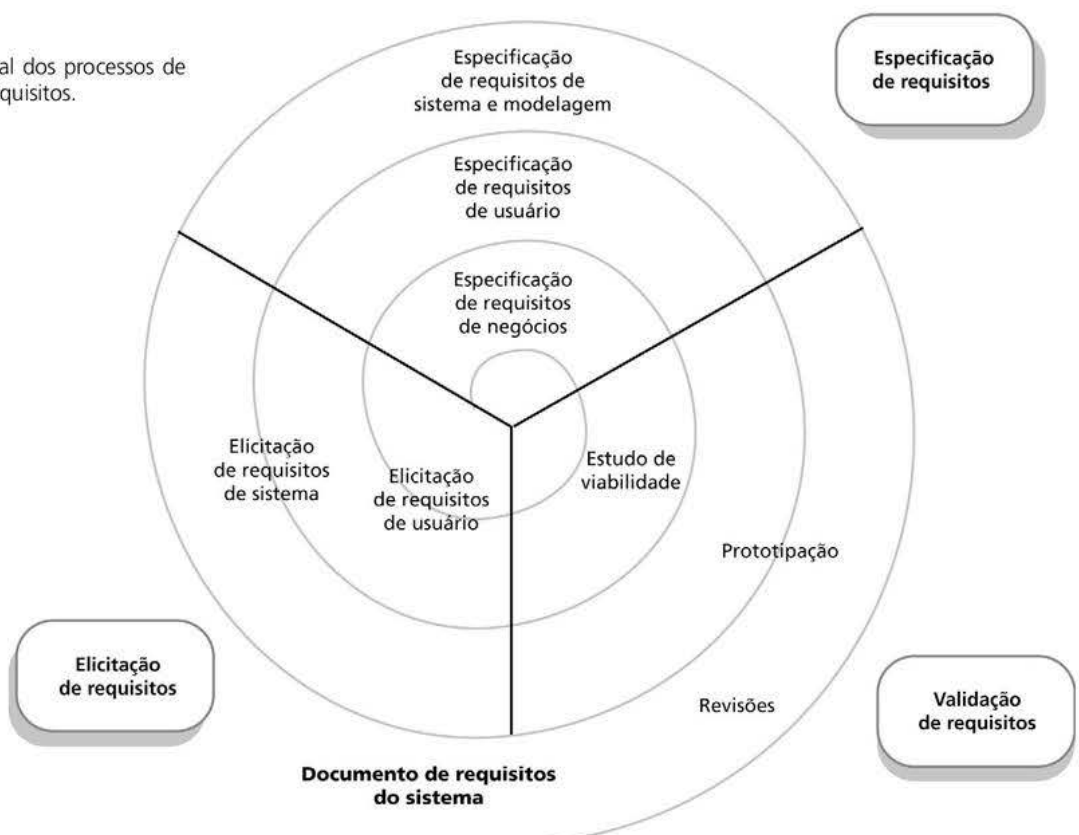


Apresento uma perspectiva alternativa do processo de engenharia de requisitos na Figura 7.2. Ela mostra o processo como uma atividade de três estágios, na qual as atividades estão organizadas como um processo iterativo em espiral. A quantidade de tempo e esforço dispensado para cada atividade na iteração depende do estágio do processo geral e do tipo de sistema que está sendo desenvolvido. No início do processo, a maior parte do esforço será empregada na compreensão dos requisitos de alto nível de negócios, requisitos não funcionais e requisitos de usuário. Próximo ao fim do processo, nas partes mais externas da espiral, um esforço maior será dedicado à engenharia de requisitos e à modelagem de sistema.

O modelo em espiral acomoda abordagens de desenvolvimento em que os requisitos são desenvolvidos para diferentes níveis de detalhes. O número de iterações na espiral pode variar, de modo que pode-se sair da espiral depois que alguns ou todos os requisitos de usuário tiverem sido elicitados. Se a atividade de prototipação apresentada sob a validação de

Figura 7.2

Modelo em espiral dos processos de engenharia de requisitos.



requisitos for ampliada para incluir o desenvolvimento iterativo, conforme explicado no Capítulo 17, este modelo permitirá que os requisitos e a implementação de sistema sejam desenvolvidos em conjunto.

Algumas pessoas consideram a engenharia de requisitos como se fosse o processo de aplicação de um método estruturado de análise como, por exemplo, a análise orientada a objetos (Larman, 2002). Isso consiste na análise do sistema e no desenvolvimento de um conjunto de modelos gráficos de sistemas, como modelos de casos de uso, que servem como uma especificação de sistema. O conjunto de modelos descreve o comportamento do sistema e inclui anotações com descrições adicionais de informações como, por exemplo, o desempenho ou a confiabilidade necessária.

Embora os métodos estruturados possuam um papel a desempenhar no processo de engenharia de requisitos, existe muito mais sobre engenharia de requisitos do que é abordado nesses métodos. A elicitação de requisitos, em particular, é uma atividade centrada em pessoas, e as pessoas não gostam das restrições impostas pelos modelos rígidos de sistema. Neste capítulo, enfoco abordagens gerais sobre engenharia de requisitos e abordo os métodos estruturados, e os modelos de sistema no Capítulo 8.

7.1 Estudos de viabilidade

Em todos os sistemas novos, o processo de engenharia de requisitos deve começar com um estudo de viabilidade. A entrada para o estudo de viabilidade consiste de um conjunto preliminar de requisitos de negócios, um esboço da descrição do sistema e como o sistema pretende apoiar os processos de negócios. Os resultados do estudo de viabilidade devem estar em um relatório que recomenda se vale a pena ou não prosseguir com os processos de engenharia de requisitos e de desenvolvimento do sistema.

Um estudo de viabilidade é um estudo breve e focalizado que procura responder a uma série de questões:

1. O sistema contribui para os objetivos gerais da organização?
2. O sistema pode ser implementado com tecnologia atual e dentro das restrições definidas de custo e prazo?
3. O sistema pode ser integrado a outros sistemas já implantados?

A questão de se o sistema contribui ou não para os objetivos da empresa é crítica. Se um sistema não apóia esses objetivos, ele não tem valor real para a empresa. Apesar de esse fato parecer óbvio, muitas organizações desenvolvem sistemas que não contribuem para seus objetivos, porque elas não têm um perfil claro desses objetivos, pois falham em definir os requisitos de negócios para o sistema, ou porque outros fatores políticos ou organizacionais influenciam na aquisição do sistema. Embora isso não seja discutido explicitamente, o estudo de viabilidade deve ser parte da fase de Concepção do Rational Unified Process, conforme explicado no Capítulo 4.

A realização de um estudo de viabilidade envolve a avaliação de informações, sua coleta e a elaboração de um relatório. A avaliação de informações identifica as informações necessárias para responder às três questões apresentadas anteriormente. Após a identificação das informações, é necessário falar com as fontes de informações para descobrir as respostas a essas perguntas. Alguns exemplos de possíveis questões que podem ser levantadas são:

1. Como a organização se comportaria se esse sistema não fosse implementado?
2. Quais são os problemas com os processos atuais e como o novo sistema ajudaria a reduzir esses problemas?
3. Qual será a contribuição direta do sistema para os objetivos e requisitos da empresa?
4. As informações podem ser transferidas e recebidas de outros sistemas da organização?
5. O sistema requer tecnologia que ainda não foi usada na organização?
6. O que deve ser apoiado pelo sistema e o que não precisa ser apoiado?

Em um estudo de viabilidade, você pode consultar fontes de informações como os gerentes de departamentos em que o sistema será usado, engenheiros de software familiarizados com o tipo de sistema proposto, especialistas em tecnologia e usuários finais do sistema. Normalmente, você deve tentar concluir um estudo de viabilidade em duas ou três semanas.

Após obter as informações, elabore o relatório de estudo de viabilidade. Você deve fazer uma recomendação de se o desenvolvimento do sistema deve ou não prosseguir. No relatório pode propor mudanças de escopo, orçamento e prazo e sugerir requisitos de alto nível adicionais para o sistema.

7.2 Elicitação e análise de requisitos

O próximo estágio do processo de engenharia de requisitos é a elicitação e análise de requisitos. Nessa atividade, os engenheiros de software trabalham com os clientes e os usuários finais do sistema para aprender sobre o domínio da aplicação, quais serviços o sistema deve fornecer, o desempenho esperado do sistema, restrições de hardware etc.

A elicitação e análise de requisitos pode envolver várias pessoas de uma organização. O termo *stakeholder* é usado para se referir a qualquer pessoa ou grupo afetado pelo sistema, direta ou indiretamente. Os stakeholders incluem os usuários finais que interagem com o sistema e todo pessoal na organização que possa ser afetado por sua instalação. Outros stakeholders no sistema podem ser os engenheiros que estão desenvolvendo ou mantendo sistemas relacionados, gerentes de negócios, especialistas do domínio e representantes do sindicato.

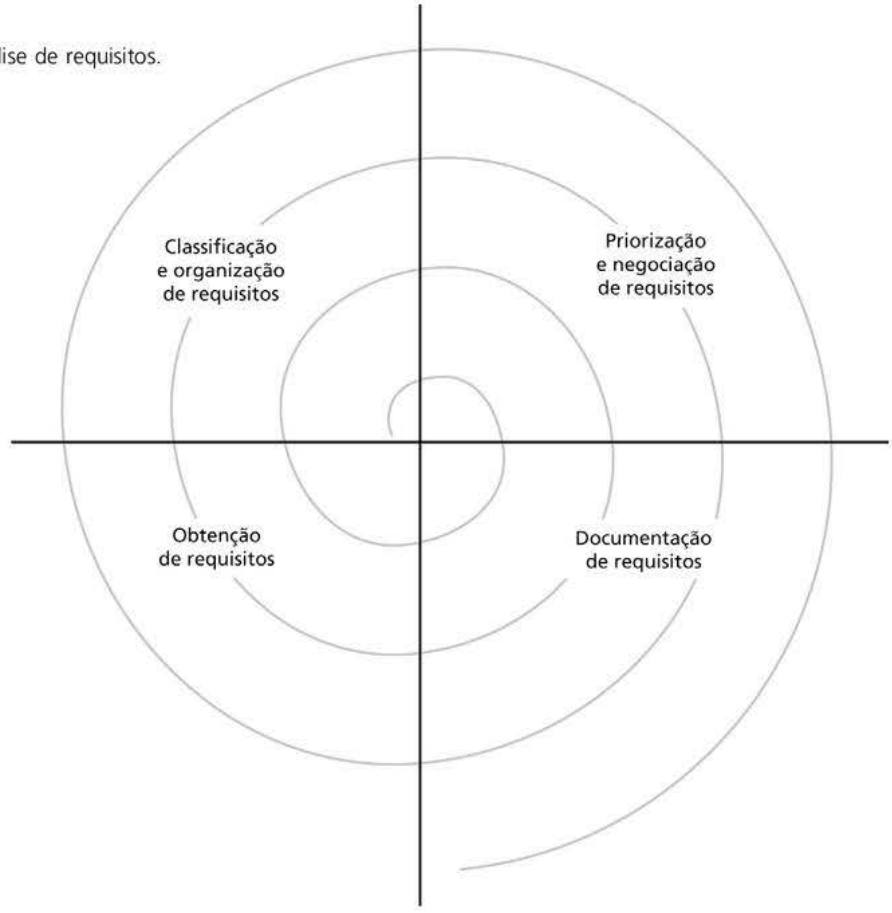
A elicitação e a compreensão dos requisitos dos stakeholders são difíceis devido a várias razões:

1. Os stakeholders freqüentemente não sabem o que querem do sistema de computador a não ser em termos gerais. Eles podem achar difícil articular o que desejam que o sistema faça ou fazem pedidos não realistas, pois ignoram o custo de seus requisitos.
2. Os stakeholders expressam os requisitos naturalmente em seus próprios termos e com o conhecimento implícito de seu trabalho. Os engenheiros de requisitos, sem experiência no domínio do cliente, devem entender esses requisitos.
3. Diferentes stakeholders possuem diferentes requisitos, expressos de diferentes formas. Os engenheiros de requisitos precisam considerar todas as fontes potenciais de requisitos e descobrir pontos em comum e conflitos.
4. Fatores políticos podem influenciar os requisitos do sistema. Por exemplo, os gerentes podem solicitar requisitos específicos do sistema que aumentarão sua influência na organização.
5. O ambiente econômico e de negócios sobre o qual a análise é realizada é dinâmico. Ele muda inevitavelmente durante o processo de análise. Portanto, a importância de determinado requisito pode mudar. Novos requisitos podem surgir de novos stakeholders que não haviam sido consultados anteriormente.

Um modelo de processo bastante genérico de elicitação e análise de requisitos é apresentado na Figura 7.3. Cada organização terá sua própria versão ou modelo desse modelo geral, dependendo de fatores locais, como o nível de conhecimento da equipe, o tipo do sistema que está sendo desenvolvido e os padrões usados. Novamente, você pode pensar nessas atividades como uma espiral, de modo que as atividades se intercalem à medida que o processo progrida da parte interna da espiral para a externa.

Figura 7.3

Processo de elicitação e análise de requisitos.



As atividades de processo são:

1. *Obtenção de requisitos*. É o processo de interação com os stakeholders no sistema para coletar seus requisitos. Os requisitos de domínio são também descobertos durante essa atividade, provenientes dos stakeholders e da documentação.
2. *Classificação e organização de requisitos*. Esta atividade envolve a coleção de requisitos não estruturados, agrupa os requisitos relacionados e os organiza em conjuntos coerentes.
3. *Priorização e negociação de requisitos*. Inevitavelmente, quando vários stakeholders participam do processo, os requisitos serão conflitantes. Esta atividade está relacionada à priorização de requisitos, à procura e à resolução de conflitos de requisitos por meio da negociação.
4. *Documentação de requisitos*. Os requisitos são documentados e colocados na próxima volta da espiral. Podem ser produzidos documentos de requisitos formais ou informais.

A Figura 7.3 mostra que a elicitação e análise de requisitos é um processo iterativo com realimentação contínua de cada atividade para outras atividades. O ciclo do processo começa com a obtenção de requisitos e termina com a documentação de requisitos. O entendimento dos requisitos pelo analista aumenta a cada volta do ciclo.

Neste capítulo, enfoco principalmente a descoberta de requisitos e as diversas técnicas desenvolvidas para dar apoio a essas atividades. A classificação e a organização de requisitos estão relacionadas principalmente com a identificação da sobreposição de requisitos dos diferentes stakeholders e o agrupamento de requisitos relacionados. O modo mais comum de agrupamento de requisitos é usar um modelo de arquitetura de sistema para identificar os subsistemas e associar os requisitos a cada subsistema. Isso enfatiza que a engenharia de requisitos e o projeto de arquitetura nem sempre podem estar separados.

Inevitavelmente, os stakeholders possuem visões diferentes sobre a importância e a prioridade dos requisitos e, algumas vezes, essas visões entram em conflito. Durante o processo, você deve organizar negociações freqüentes com os stakeholders, de modo que os compromissos possam ser atingidos. É impossível satisfazer completamente a todos os stakeholders, mas se alguns deles perceberem que suas visões não foram consideradas de maneira apropriada, eles podem tentar boicotar intencionalmente o processo de engenharia de requisitos (RE — Requirements Engineering).

No estágio de documentação, os requisitos elicitados são documentados de modo que possam ser usados para auxiliar as próximas obtenções de requisitos. Nesse estágio, uma versão inicial dos requisitos do sistema pode ser produzida, mas terá seções faltantes e requisitos incompletos. Como alternativa, os requisitos podem ser documentados como tabelas em um documento ou em cartões. Escrever os requisitos em cartões (abordagem usada na *extreme programming*) pode ser muito eficiente, pois os stakeholders podem manusear, alterar e organizar os cartões com facilidade.

7.2.1 Obtenção de requisitos

A obtenção de requisitos é o processo que reúne informações sobre o sistema proposto e os existentes para obter os requisitos de usuário e de sistema com base nessas informações. As fontes de informações, durante a fase de obtenção de requisitos, incluem documentação, stakeholders de sistema e especificações de sistemas similares. A interação com os stakeholders ocorre por meio de entrevistas e observações, podendo ser usados cenários e protótipos para auxiliar na obtenção dos requisitos. Nesta seção, explico uma abordagem que ajuda a assegurar uma cobertura ampla de stakeholders durante a obtenção de requisitos e descrevo técnicas que incluem entrevistas, cenários e etnografia. Outras técnicas de obtenção de requisitos que podem ser usadas incluem métodos estruturados de análise, abordados no Capítulo 8, e prototipação de sistema, abordada no Capítulo 17.

Os stakeholders variam de usuários finais do sistema a gerentes e envolvidos externos, como regulamentadores que certificam a aceitação do sistema. Por exemplo, os stakeholders em um sistema de caixa eletrônico bancário incluem:

1. *Clientes atuais do banco* que recebem serviços do sistema.
2. *Representantes de outros bancos* que têm acordos recíprocos que permitem usar os caixas eletrônicos uns dos outros.
3. *Gerentes de agências bancárias* que obtêm informações de gerenciamento do sistema.
4. *Pessoal de atendimento nas agências bancárias* envolvidos na operação diária do sistema.
5. *Administradores de banco de dados* responsáveis pela integração do sistema com o banco de dados dos clientes do banco.
6. *Gerentes de proteção bancária* que devem assegurar que o sistema não seja exposto a riscos de proteção.
7. *Departamento de marketing do banco* provavelmente interessado em usar o sistema como meio de marketing do banco.
8. *Engenheiros de manutenção de hardware e software* que são responsáveis pela manutenção e atualização de hardware e software.

9. *Reguladores nacionais de bancos* responsáveis por assegurar que o sistema esteja em conformidade com os regulamentos bancários.

Além dos stakeholders no sistema, vimos anteriormente que os requisitos podem ser provenientes do domínio de aplicação e de outros sistemas que interagem com o sistema que está sendo especificado. Tudo isso deve ser considerado durante o processo de elicitação de requisitos.

Essas fontes de requisitos (stakeholders, domínio, sistemas) podem ser representadas como pontos de vista do sistema, em que cada ponto de vista apresenta um subconjunto de requisitos do sistema. Cada ponto de vista fornece uma perspectiva nova do sistema, mas essas perspectivas não são completamente independentes — elas geralmente se sobrepõem de modo que haja requisitos comuns.

Pontos de vista

As abordagens orientadas a pontos de vista para engenharia de requisitos (Mullery, 1979; Finkelstein et al., 1992; Kotonya e Sommerville, 1992; Kotonya e Sommerville, 1996) organizam o processo de elicitação e os próprios requisitos usando pontos de vista. Um ponto forte da análise orientada a pontos de vista é que ela reconhece várias perspectivas e fornece um framework para descobrir conflitos nos requisitos propostos por diferentes stakeholders.

Os pontos de vista podem ser usados como um meio de classificação de stakeholders e de outras fontes de requisitos. Existem três tipos genéricos de pontos de vista:

1. *Pontos de vista de interação*: representam pessoas ou outros sistemas que interagem diretamente com o sistema. No sistema de caixa eletrônico bancário, exemplos de pontos de vista de interação são os clientes do banco e o banco de dados de contas bancárias.
2. *Pontos de vista indiretos*: representam os stakeholders que não usam o sistema diretamente, mas que influenciam os requisitos de alguma forma. No sistema de caixa eletrônico bancário, exemplos de pontos de vista indiretos são a gerência do banco e o pessoal de proteção do banco.
3. *Pontos de vista de domínio*: representam características e restrições de domínio que influenciam os requisitos de sistema. No sistema de caixa eletrônico bancário, um exemplo de um ponto de vista de domínio são os padrões desenvolvidos para comunicações entre os bancos.

Tipicamente, esses pontos de vista fornecem diferentes tipos de requisitos. Os pontos de vista de interação fornecem requisitos de sistema detalhados que abrangem as características e as interfaces do sistema. Os pontos de vista indiretos são os que mais provavelmente fornecem requisitos e restrições organizacionais de alto nível. Os pontos de vista de domínio fornecem normalmente as restrições de domínio que se aplicam ao sistema.

A identificação inicial dos pontos de vista relevantes ao sistema pode ser difícil algumas vezes. Para ajudar nesse processo, você deve tentar identificar tipos mais específicos de pontos de vista:

1. Fornecedores e receptores de serviços do sistema.
2. Sistemas que devem interfacear diretamente com o sistema que está sendo especificado.
3. Regulamentos e padrões que se aplicam ao sistema.
4. Fontes de requisitos de negócios e não funcionais do sistema.
5. Pontos de vista de engenharia que refletem os requisitos de pessoas que devem desenvolver, gerenciar e manter o sistema.
6. Pontos de vista de marketing e outros pontos de vista que geram requisitos de características do produto esperadas pelos clientes e como o sistema deve refletir a imagem externa da organização.

Quase todos os sistemas organizacionais devem interoperar com outros sistemas na organização. Quando um novo sistema é planejado, as interações com outros sistemas devem ser planejadas. As interfaces oferecidas por esses outros sistemas já foram projetadas. Elas podem incluir requisitos e restrições sobre o novo sistema. Além disso, novos sistemas podem precisar estar em conformidade com regulamentos e padrões preexistentes que restringem os requisitos do sistema.

Conforme expliquei anteriormente neste capítulo, você deve identificar os requisitos de negócio e não funcionais de alto nível no início do processo de RE. As fontes desses requisitos podem ser pontos de vista úteis em um processo mais detalhado. Elas devem ser capazes de expandir e desenvolver os requisitos de alto nível gerando requisitos de sistema mais específicos.

Os pontos de vista de engenharia podem ser importantes por duas razões. Primeiro, os engenheiros que desenvolvem o sistema podem ter experiência com sistemas similares e ser capazes de sugerir requisitos levando em conta essa experiência. Segundo, o pessoal técnico que deve gerenciar e manter o sistema pode ter requisitos que irão ajudar a simplificar o apoio ao sistema. Os requisitos de gerenciamento de sistema são cada vez mais importantes, pois os custos de gerenciamento constituem uma porção cada vez maior dos custos totais no tempo de vida do sistema.

Finalmente, os pontos de vista que fornecem requisitos podem ser provenientes dos departamentos de marketing ou de negócios externos em uma organização. Isso é especialmente verdadeiro em sistemas baseados na Web, mais especificamente sistemas de e-commerce e softwares comerciais. Os sistemas baseados na Web devem apresentar uma imagem favorável da organização, assim como oferecer funcionalidade ao usuário. No tocante a produtos de software, o departamento de marketing deve conhecer quais características tornarão o sistema mais atraente aos potenciais compradores.

Para qualquer sistema não trivial, existe um grande número de possíveis pontos de vista, e é praticamente impossível elicitar os requisitos baseado em todos eles. Portanto, é importante que você organize e estruture os pontos de vista em uma hierarquia. Os pontos de vista do mesmo ramo provavelmente compartilharão requisitos comuns.

Como ilustração, considere a hierarquia de pontos de vista apresentada na Figura 7.4. É um diagrama relativamente simples de pontos de vista que podem ser consultados para derivar requisitos para o sistema LIBSYS. Você pode observar que a classificação dos pontos de vista de interação, indiretos e de domínio ajuda a identificar fontes de requisitos independente dos usuários imediatos do sistema.

Depois que os pontos de vista forem identificados e estruturados, você pode tentar identificar os pontos de vista mais importantes e iniciar com eles a obtenção de requisitos do sistema.

Entrevista

Entrevistas formais ou informais com os stakeholders no sistema fazem parte da maioria dos processos de engenharia de requisitos. Nessas entrevistas, a equipe de engenharia de requisitos formula questões para os stakeholders sobre o sistema que eles usam e o sistema a ser desenvolvido. Os requisitos são derivados das respostas a essas questões. As entrevistas podem ser de dois tipos:

1. Entrevistas fechadas, nas quais o stakeholder responde a um conjunto de perguntas predefinidas.
2. Entrevistas abertas, nas quais não existe um roteiro predefinido. A equipe de engenharia de requisitos explora vários assuntos com os stakeholders no sistema e, assim, desenvolve uma compreensão maior de suas necessidades.

Na prática, as entrevistas com os stakeholders são, geralmente, uma combinação desses tipos. As respostas a algumas perguntas podem levar a outros assuntos discutidos de maneira menos estruturada. As discussões completamente abertas raramente funcionam bem; a maioria das entrevistas requer algumas perguntas como ponto de partida e para manter o foco no sistema a ser desenvolvido.

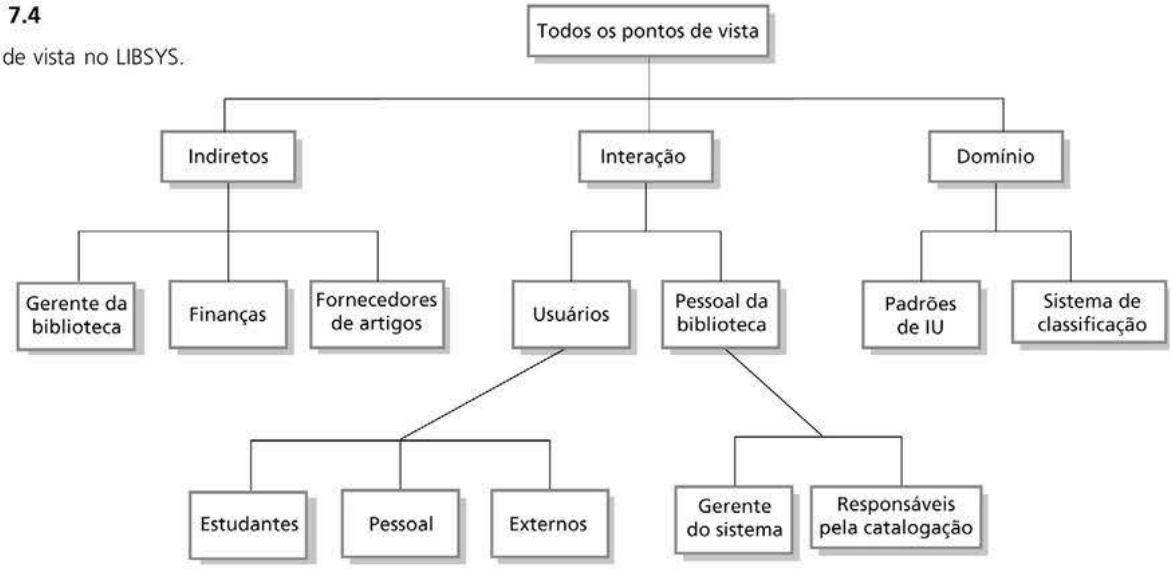
As entrevistas são úteis para obter um entendimento geral sobre o que os stakeholders fazem, como eles podem interagir com o sistema e as dificuldades que enfrentam com os sistemas atuais. As pessoas gostam de falar sobre seu trabalho e, normalmente, ficam felizes em participar de entrevistas. No entanto, as entrevistas não são tão úteis para compreender os requisitos do domínio da aplicação.

É difícil elicitar o conhecimento de domínio durante as entrevistas por dois motivos:

1. Todos os especialistas de domínio usam terminologia e jargões específicos. É impossível para eles explicar os requisitos de domínio sem usar essa terminologia. Eles em geral usam a terminologia de maneira precisa e específica, o que facilmente provoca mal-entendidos por parte dos engenheiros de requisitos.

Figura 7.4

Pontos de vista no LIBSYS.



2. Alguns conhecimentos de domínio são tão familiares aos stakeholders que são considerados difíceis de explicar ou considerados tão fundamentais que não vale a pena mencioná-los. Por exemplo, para uma bibliotecária, não há necessidade de dizer que todas as aquisições são catalogadas antes de serem colocadas na biblioteca. Contudo, isso pode não ser óbvio para o entrevistador e, portanto, não é levado em consideração nos requisitos.

A entrevista não é uma técnica eficiente para elicitação de conhecimentos sobre os requisitos e as restrições organizacionais, pois existem relacionamentos sutis de poder e influência entre os stakeholders na organização. As estruturas organizacionais públicas raramente coincidem com a realidade da tomada de decisões na organização, mas os entrevistados podem não querer revelar a estrutura real, em lugar da teórica, a um estranho. Em geral, a maioria das pessoas é relutante em discutir questões políticas e organizacionais que podem afetar os requisitos.

Os entrevistadores eficientes possuem duas características:

1. Possuem mente aberta, evitam idéias preconcebidas sobre os requisitos e querem ouvir os stakeholders. Se o stakeholder propuser requisitos surpreendentes, os entrevistadores estão dispostos a mudar de idéia sobre o sistema.
2. Induzem os entrevistados a iniciar as discussões com uma questão, uma proposta de requisitos ou sugerindo um trabalho conjunto em um protótipo do sistema. Dizer para as pessoas 'conte-me o que você quer' provavelmente trará informações úteis como resultado. A maioria das pessoas considera mais fácil falar em um contexto definido do que em termos gerais.

As informações obtidas das entrevistas complementam outras informações sobre o sistema obtidas de documentos, observações de usuários etc. Às vezes, independentemente das informações de documentos, as entrevistas podem ser as únicas fontes de informações sobre os requisitos do sistema. No entanto, apenas com as entrevistas pode haver a perda de informações essenciais; assim, essa técnica deve ser usada junto com outras para a elicitação de requisitos.

Cenários

As pessoas geralmente consideram mais fácil relatar exemplos da vida real do que abstrair descrições. Elas podem compreender e criticar um cenário de como interagiriam com um sistema de software. Os engenheiros de requisitos podem usar as informações obtidas nessa discussão para elaborar os requisitos reais do sistema.

Os cenários podem ser particularmente úteis para adicionar detalhes a um esboço da descrição de requisitos. Eles são descrições de exemplos das sessões de interação. Cada cenário abrange uma ou mais interações possíveis. Diversos tipos de cenários foram desenvolvidos, cada um dos quais fornecendo diferentes tipos de informações sobre o sistema em diferentes níveis de detalhamento. O uso de cenários para descrever requisitos é parte integrante dos métodos ágeis, como a *extreme programming*, que explico no Capítulo 17.

O cenário começa com um esboço da interação e, durante a elicitação, os detalhes são adicionados para criar uma descrição completa dessa interação. De maneira geral, um cenário deve incluir:

1. Uma descrição do que os usuários esperam do sistema no início do cenário
2. Uma descrição do fluxo normal de eventos no cenário
3. Uma descrição do que pode dar errado e como isso é tratado
4. Informações sobre outras atividades que podem ocorrer simultaneamente
5. Uma descrição do estado de sistema no fim do cenário

A elicitação baseada em cenários pode ser realizada informalmente — os engenheiros de requisitos trabalham com os stakeholders para identificar cenários e captar os detalhes desses cenários. Os cenários podem ser escritos na forma de textos, complementados por diagramas, imagens de computador etc. Como alternativa, pode ser adotada uma abordagem mais estruturada, como cenários de eventos ou casos de uso.

Como exemplo de um cenário simples em texto, considere como um usuário do sistema LIBSYS de biblioteca pode usar o sistema. O cenário é mostrado no Quadro 7.1. O usuário deseja imprimir uma cópia para uso pessoal de um artigo de uma revista médica. Essa revista permite cópias gratuitas de artigos disponíveis para assinantes, mas os não-assinantes devem pagar uma taxa por artigo. O usuário conhece o artigo, seu título e a data da publicação.

Casos de uso

Os casos de uso constituem uma técnica baseada em cenários para elicitação de requisitos e foram introduzidos inicialmente no método Objectory (Jacobsen, et al., 1993). Eles se tornaram uma característica fundamental da notação UML para descrição de modelos de sistema orientado a objetos. Em sua forma mais simples, um caso de uso identifica o tipo da interação e os agentes envolvidos. Por exemplo, a Figura 7.5 mostra o caso de uso de alto nível do recurso de impressão de artigos no LIBSYS, descrito no Quadro 7.1.

Quadro 7.1

Cenário para download de artigo no LIBSYS.

Hipótese inicial: O usuário se conectou ao sistema LIBSYS e localizou a revista que contém a cópia do artigo.

Normal: O usuário seleciona o artigo a ser copiado. O sistema solicita que o usuário forneça as informações de assinante da revista ou indique uma forma de pagamento pelo artigo. O pagamento pode ser feito por meio de cartão de crédito ou com a informação de um número de conta da organização.

É solicitado, depois, que o usuário preencha um formulário de direitos autorais com os detalhes da transação e o envie ao sistema LIBSYS.

O formulário de direitos autorais é verificado e, caso aprovado, a versão do artigo em PDF é baixada na área de trabalho do LIBSYS no computador do usuário e este é avisado de que o artigo está disponível. É solicitado que o usuário selecione uma impressora, e uma cópia do artigo é impressa. Se o artigo estiver marcado como 'apenas para impressão', este será apagado do sistema do usuário após o término da impressão.

O que pode dar errado: O usuário pode não preencher o formulário de direitos autorais corretamente. Nesse caso, o formulário deverá ser reapresentado ao usuário para correção. Se o formulário reapresentado ainda estiver incorreto, a solicitação do usuário para o artigo será rejeitada.

O pagamento pode ser rejeitado pelo sistema; nesse caso, a solicitação do usuário para o artigo será rejeitada.

O download do artigo pode falhar, o que faz com que o sistema tente novamente até que a operação seja bem-sucedida ou que o usuário termine a sessão.

Pode não ser possível imprimir o artigo. Se o artigo não estiver marcado como 'apenas para impressão', ele será mantido na área de trabalho do LIBSYS. Caso contrário, o artigo será apagado e o custo do artigo será debitado na conta do usuário.

Outras atividades: Downloads simultâneos de outros artigos.

Estado de sistema após o término: O usuário estará conectado. O artigo baixado teria sido apagado da área de trabalho do LIBSYS caso estivesse marcado como 'apenas para impressão'.

A Figura 7.5 mostra os elementos essenciais da notação de caso de uso. Os agentes no processo são representados como bonecos, e cada classe de interação é representada como uma elipse identificada. O conjunto de casos de uso representa todas as possíveis interações a serem representadas nos requisitos de sistema. A Figura 7.6 desenvolve o exemplo do LIBSYS e mostra outros casos de uso nesse ambiente.

Algumas vezes, existe confusão sobre se um caso de uso é um cenário em si ou, conforme sugerido por Fowler (Fowler e Scott, 1997), um caso de uso engloba um conjunto de cenários, sendo cada cenário um encadeamento isolado ao longo do caso de uso. Se um cenário incluir vários encadeamentos, existirá um cenário para interação normal e mais cenários para cada possível exceção.

Os casos de uso identificam as interações individuais com o sistema. Eles podem ser documentados por meio de texto ou de links com os modelos UML que desenvolvem o cenário mais detalhadamente. Os diagramas de seqüência (apresentados no Capítulo 6) são freqüentemente usados para adicionar informações ao caso de uso. Os diagramas de seqüência mostram os agentes envolvidos na interação, os objetos com os quais interagem e as operações associadas a esses objetos.

Como ilustração disto, a Figura 7.7 mostra as interações envolvidas no uso do LIBSYS para baixar e imprimir um artigo. Na Figura 7.7, existem quatro objetos de classes — Artigo, Formulário, Área de Trabalho e Impressora — envolvidos na interação. A seqüência de ações ocorre de cima para baixo, e os rótulos sobre as setas entre os agentes e os objetos indicam os nomes das operações. Essencialmente, a solicitação de um artigo pelo usuário dispara uma requisição para um formulário de direitos autorais. Quando o usuário concluir o preenchimento do formulário, o artigo será baixado e enviado para a impressora. Quando a impressão terminar, o artigo será apagado da área de trabalho do LIBSYS.

A UML é um padrão real para a modelagem orientada a objetos e, portanto, os casos de uso e a elicitação baseada em casos de uso têm sido cada vez mais usados para elicitação de requisitos. Outros tipos de modelos UML são explicados no Capítulo 8, que aborda a modelagem de sistema, e no Capítulo 14, que aborda o projeto orientado a objetos.

Cenários e casos de uso são técnicas eficazes para elicitação de requisitos segundo pontos de vista de interação, em que cada tipo de interação pode ser representado como um caso de uso. Eles também podem ser usados em conjunto com alguns pontos de vista indiretos, sendo que esses pontos de vista recebem alguns resultados (como um relatório gerencial)

Figura 7.5

Caso de uso simples para impressão de artigos.

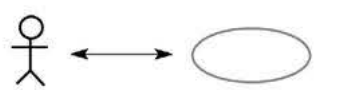


Figura 7.6

Casos de uso para o sistema de biblioteca.

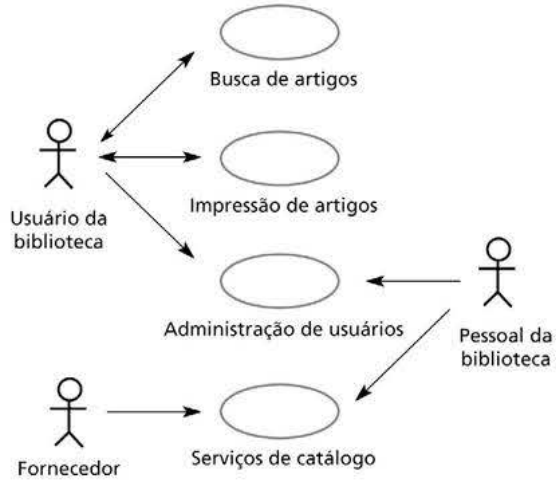
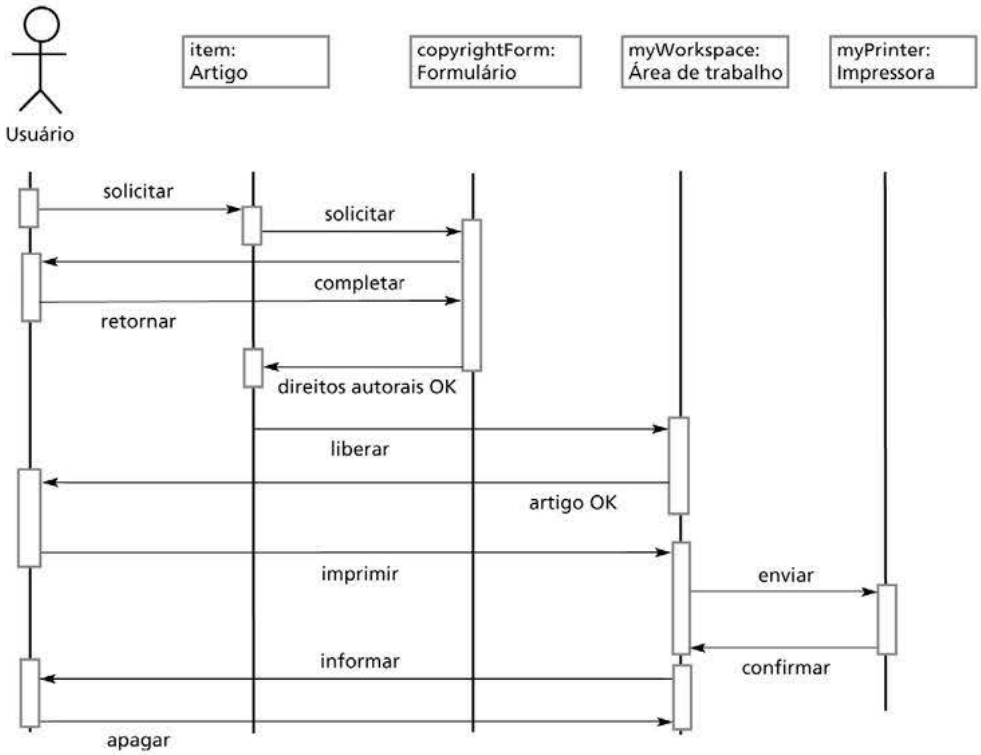


Figura 7.7

Interações do sistema para impressão de artigos.



do sistema. Contudo, como eles enfocam as interações, não são eficazes para elicitar restrições ou requisitos de negócios e não funcionais de alto nível com base nos pontos de vista indiretos ou para obter requisitos de domínio.

7.2.2 Etnografia

Os sistemas de software não existem isoladamente — eles são usados em um contexto social e organizacional e os requisitos do sistema de software podem ser derivados ou restringidos por esse contexto. Geralmente, satisfazer esses requisitos sociais e organizacionais é importante para o sucesso do sistema. Uma razão pela qual vários sistemas de software são entregues, mas nunca usados, é que eles não dão a importância devida a esses requisitos.

Etnografia é uma técnica de observação que pode ser usada para compreender os requisitos sociais e organizacionais. Um analista se insere no ambiente de trabalho onde o sistema será usado. Ele observa o trabalho do dia-a-dia e anota as tarefas reais nas quais os participantes estão envolvidos. O valor da etnografia está na ajuda que presta aos analistas para descobrir os requisitos implícitos de sistema que refletem os processos reais, e não os formais, com os quais as pessoas estão envolvidas.

Figura 7.8

Etnografia e prototipação para análise de requisitos.

As pessoas freqüentemente consideram muito difícil articular detalhes de seu trabalho pois isso é secundário para elas. Elas compreendem seu próprio trabalho, mas podem não compreender seu relacionamento com o trabalho de outros na organização. Os fatores sociais e organizacionais, que afetam o trabalho, mas que não são óbvios para as pessoas, podem somente se tornar claros quando examinados por um observador imparcial.

Suchman (Suchman, 1987) usou a etnografia para estudar o trabalho em escritório e concluiu que as práticas reais de trabalho eram mais ricas, mais complexas e mais dinâmicas do que os simples modelos considerados pelos sistemas de automação de escritório. A diferença entre o trabalho suposto e o real é a razão mais importante pela qual os sistemas de escritório não tiveram efeito significativo na produtividade. Outros estudos de etnografia para a compreensão dos requisitos de sistema incluíram estudos sobre controle de tráfego aéreo (Bentley, et al., 1992; Hughes, et al., 1993), salas de controle de metrô (Heath e Luff, 1992), sistemas financeiros e várias atividades de projeto (Heath, et al., 1993; Hughes, et al., 1994). Em minhas pesquisas, investiguei métodos de integração de etnografia no processo de engenharia de software, ligando-o aos métodos de engenharia de requisitos (Viller e Sommerville, 1999; Viller e Sommerville, 1998; Viller e Sommerville, 2000) e pela documentação de padrões de interação em sistemas cooperativos (Martin, et al., 2001; Martin, et al., 2002; Martin e Sommerville, 2004).

A etnografia é particularmente eficaz para descobrir dois tipos de requisitos:

1. *Requisitos derivados da maneira como as pessoas realmente trabalham* em vez da maneira pela qual as definições de processo dizem que elas deveriam trabalhar. Por exemplo, os controladores de tráfego aéreo podem desligar um sistema de alerta de conflito de aeronaves que detecta rotas de vôo em colisão, embora os procedimentos normais de controle especifiquem que ele deve ser usado. Sua estratégia de controle é projetada para assegurar que essas aeronaves sejam afastadas antes que ocorram problemas e os controladores consideram que o alarme de alerta os distrai de seu trabalho.
2. *Requisitos derivados da cooperação e do conhecimento das atividades de outras pessoas.* Por exemplo, os controladores de tráfego aéreo podem usar o conhecimento que têm do trabalho de outros controladores para prever o número de aeronaves que entrarão em seu setor de controle. Depois eles modificam suas estratégias de controle, dependendo da sobrecarga prevista. Portanto, um sistema automatizado de controle de tráfego aéreo deve permitir que os controladores em um setor tenham alguma visibilidade do trabalho em setores adjacentes.

A etnografia pode ser combinada com a prototipação (Figura 7.8). A etnografia informa o desenvolvimento do protótipo de tal modo que poucos ciclos de refinamento de protótipo sejam necessários. Além disso, a prototipação enfoca a etnografia, identificando os problemas e as questões que podem, assim, ser discutidos com o etnógrafo. O etnógrafo deve, então, procurar as respostas a estas questões durante a próxima fase do estudo de sistema (Sommerville, et al., 1993).

Os estudos de etnografia podem revelar detalhes importantes do processo freqüentemente ignorados por outras técnicas de elicitação de requisitos. No entanto, devido a seu foco no usuário final, essa abordagem não é apropriada para obter os requisitos organizacionais ou de domínio. Os estudos etnográficos nem sempre podem identificar novas características a serem acrescentadas a um sistema. A etnografia não é, portanto, uma abordagem completa para elicitação e deve ser usada para complementar outras abordagens, como a análise de casos de uso.

7.3 Validação de requisitos

A validação de requisitos dedica-se a mostrar que os requisitos realmente definem o sistema que o usuário deseja. A validação de requisitos se sobrepõe à análise; está relacionada à descoberta de problemas com os requisitos. A validação de requisitos é importante porque os erros em um documento de requisitos podem levar a custos excessivos de retrabalho quando são descobertos durante o desenvolvimento ou depois que o sistema está em operação. O custo de correção de um problema de requisitos, fazendo uma mudança de sistema, é muito maior do que a correção de erros de projeto e de codificação. A razão disso é que uma mudança de requisitos significa geralmente que o projeto e a implementação do sistema devem também ser mudados e o sistema deve ser novamente testado.

Durante o processo de validação de requisitos, devem ser realizadas verificações nos requisitos do documento de requisitos. Essas verificações incluem:

1. *Verificações de validade*. Um usuário pode pensar que um sistema é necessário para desempenhar determinadas funções. Contudo, mais estudos e análises podem identificar que funções adicionais e diferentes são necessárias. Os sistemas têm diversos stakeholders com necessidades diferentes e qualquer conjunto de requisitos é, inevitavelmente, um compromisso da comunidade de stakeholders.
2. *Verificações de consistência*. Os requisitos em um documento não devem ser conflitantes. Isso significa que não devem existir restrições ou descrições contraditórias para a mesma função do sistema.
3. *Verificações de completeza*. O documento de requisitos deve incluir requisitos que definam todas as funções e as restrições desejadas pelo usuário do sistema.
4. *Verificações de realismo*. Usando o conhecimento da tecnologia existente, os requisitos devem ser verificados quanto a se realmente podem ser implementados. Essas verificações também devem levar em consideração o orçamento e o prazo para o desenvolvimento do sistema.
5. *Facilidade de verificação*. Para reduzir o potencial de divergências entre cliente e fornecedor, os requisitos do sistema devem sempre ser escritos de modo que sejam verificáveis. Isso significa que você deve ser capaz de escrever um conjunto de testes que possa demonstrar que o sistema entregue atende a cada requisito especificado.

Uma série de técnicas de validação de requisitos pode ser usada em conjunto ou individualmente:

1. *Revisões de requisitos*. Os requisitos são analisados sistematicamente por uma equipe de revisores. Esse processo será explicado na próxima seção.
2. *Prototipação*. Nesta abordagem de validação, um modelo executável do sistema é apresentado para usuários finais e clientes. Eles podem experimentar o modelo para verificar se atende às suas necessidades reais. Explico a prototipação e suas técnicas no Capítulo 17.
3. *Geração de casos de teste*. Os requisitos devem ser testáveis. Se os testes dos requisitos forem criados como parte do processo de validação, eles freqüentemente revelarão problemas de requisitos. Se um teste for difícil ou impossível de ser projetado, isso significa geralmente que os requisitos serão difíceis de ser implementados e devem ser reconsiderados. O desenvolvimento de testes a partir de requisitos de usuário, antes de o código estar escrito, é uma parte integrante da *extreme programming*.

Não se deve subestimar os problemas de validação de requisitos. É difícil demonstrar que um conjunto de requisitos atende às necessidades do usuário. Os usuários devem imaginar o sistema em operação e avaliar sua adequação ao trabalho. É difícil para profissionais de informática habilidosos realizarem esse tipo de análise abstrata e é ainda mais difícil para os usuários do sistema. Como resultado, você raramente encontra todos os problemas de requisitos durante o processo de validação. É inevitável que haja mudanças de requisitos posteriores para corrigir omissões e mal-entendidos depois da aprovação do documento de requisitos.

7.3.1 Revisões de requisitos

A revisão de requisitos é um processo manual que envolve pessoas de ambas as organizações, do cliente e do fornecedor. Eles verificam o documento de requisitos em busca de anomalias e omissões. O processo de revisão pode ser gerenciado da mesma maneira que as inspeções de programa (veja o Capítulo 22). Como alternativa, ele pode ser organizado como uma atividade mais ampla, sendo que diferentes pessoas verificam diferentes partes do documento.

As revisões de requisitos podem ser informais ou formais. As revisões informais envolvem simplesmente os fornecedores, que discutem os requisitos com o maior número possível de stakeholders. É surpreendente a freqüência com que a comunicação entre os projetistas e os stakeholders termina após a elicitação e não existe confirmação de se os requisitos documentados são os que os stakeholders realmente solicitaram. Muitos problemas podem ser detectados simplesmente conversando sobre o sistema com os stakeholders antes do comprometimento de uma revisão formal.

Em uma revisão formal de requisitos, a equipe de desenvolvimento deve 'conduzir' o cliente pelos requisitos de sistema, explicando as implicações de cada requisito. A equipe de revisão deve verificar cada requisito em termos de consistência, bem como verificar os requisitos como um todo em termos de completeza. Os revisores podem também verificar:

1. *Facilidade de verificação*. O requisito, conforme estabelecido, é testável de forma realística?
2. *Facilidade de compreensão*. Os adquirentes e os usuários finais do sistema compreendem o requisito de forma apropriada?

3. *Rastreabilidade*. A origem do requisito está estabelecida claramente? Talvez seja necessário retornar à fonte do requisito para avaliar o impacto de uma mudança. A rastreabilidade é importante, pois permite que o impacto de uma mudança seja avaliado em relação ao restante do sistema. Eu explico a rastreabilidade mais detalhadamente na próxima seção.
4. *Adaptabilidade*. O requisito é adaptável? Isto é, o requisito pode ser mudado sem efeitos em grande escala sobre os outros requisitos de sistema?

Conflitos, contradições, erros e omissões nos requisitos devem ser apontados pelos revisores e registrados formalmente no relatório de revisão. É, portanto, de responsabilidade dos usuários, do adquirente de sistema e do desenvolvedor de sistema negociar uma solução para esses problemas identificados.

7.4 Gerenciamento de requisitos

Os requisitos de sistemas de software de grande porte estão sempre mudando. Um motivo para isso é que esses sistemas são geralmente desenvolvidos para lidar com problemas 'perversos' (conforme explicado no Capítulo 2). Como o problema não pode ser totalmente definido, os requisitos de software tendem a ser incompletos. Durante o processo de software, o entendimento dos stakeholders sobre o problema muda constantemente. Esses requisitos devem então evoluir para refletir essas novas visões do problema.

Além disso, depois que o sistema estiver instalado, inevitavelmente surgirão novos requisitos. É difícil para os usuários e os clientes do sistema anteciparem quais efeitos o novo sistema causará na organização. Quando os usuários adquirirem experiência com o sistema, eles descobrirão novas necessidades e prioridades:

1. Sistemas de grande porte geralmente têm uma comunidade diversificada de usuários, sendo que esses usuários têm diferentes requisitos e prioridades, os quais podem ser conflitantes ou contraditórios. Os requisitos finais do sistema constituem inevitavelmente um compromisso entre eles e, com a experiência, descobre-se freqüentemente que o equilíbrio de apoio dado aos diferentes usuários tem de ser mudado.
2. As pessoas que pagam por um sistema e os usuários de um sistema raramente são as mesmas pessoas. Os clientes do sistema impõem requisitos devido a restrições organizacionais e de orçamento. Essas restrições podem ser conflitantes com os requisitos dos usuários finais e, após a entrega, novas características podem ser incluídas para apoio do usuário, fazendo com que o sistema alcance suas metas.
3. O ambiente de negócios e técnico do sistema muda após a instalação, e essas mudanças devem se refletir no sistema. Um novo hardware pode ser introduzido, pode ser necessária uma interface com outros sistemas, as prioridades de negócios podem mudar trazendo conseqüentes mudanças no apoio ao sistema, e um nova legislação e regulamentos podem ser introduzidos devendo ser implementados no sistema.

O gerenciamento de requisitos é um processo para compreender e controlar as mudanças dos requisitos de sistema. Você precisa manter o acompanhamento dos requisitos individuais e manter as ligações entre os requisitos dependentes, de modo que seja possível avaliar o impacto das mudanças de requisitos. Você precisa estabelecer um processo formal para fazer propostas de mudança e ligá-las aos requisitos de sistema. O processo de gerenciamento de requisitos deve se iniciar assim que uma versão inicial do documento de requisitos esteja disponível, mas você deve iniciar o planejamento das mudanças de requisitos durante o processo de elicitação de requisitos.

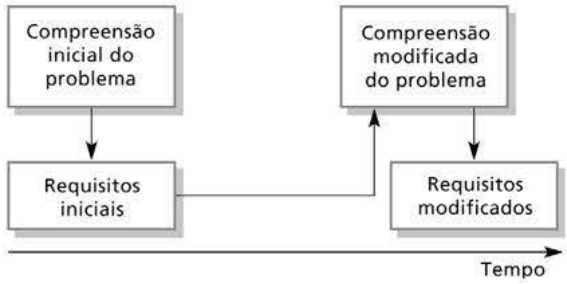
7.4.1 Requisitos permanentes e voláteis

A evolução de requisitos, durante o processo de RE e após a entrada de um sistema em operação, é inevitável. O desenvolvimento de requisitos de software enfoca as capacidades de software, objetivos da empresa e outros sistemas da empresa. À medida que a definição dos requisitos se desenvolve, uma compreensão maior das necessidades dos usuários é obtida. Isso realimenta as informações do usuário que pode, então, propor uma mudança nos requisitos (Figura 7.9). Além disso, pode levar muitos anos para especificar um sistema de grande porte. Ao longo desse tempo, o ambiente do sistema e os objetivos da empresa mudam, e os requisitos evoluem para refletir essas mudanças.

Do ponto de vista da evolução, os requisitos dividem-se em duas classes:

1. *Requisitos permanentes*. São requisitos relativamente estáveis derivados da atividade central da organização e que se relacionam diretamente ao domínio do sistema. Por exemplo, em um hospital, sempre existirão requisitos relacionados a pacientes, médicos, enfermeiros e tratamentos. Esses requisitos podem ser derivados de modelos de domínio que mostram as entidades e as relações que caracterizam um domínio de aplicação (Easterbrook, 1993; Prieto-Díaz e Arango, 1991).

Figura 7.9
Evolução de requisitos.



2. *Requisitos voláteis*. São requisitos que provavelmente irão mudar durante o processo de desenvolvimento do sistema ou depois que o sistema estiver em operação. Um exemplo seria os requisitos que resultam de políticas de saúde do governo.

Harker e outros (Harker, et al., 1993) sugeriram que os requisitos voláteis fossem divididos em cinco classes. Usando isso como base, desenvolvi a classificação mostrada na Tabela 7.1.

7.4.2 Planejamento de gerenciamento de requisitos

O planejamento é o primeiro estágio essencial no processo de gerenciamento de requisitos. O gerenciamento de requisitos é muito dispendioso. Para cada projeto, o estágio de planejamento estabelece o nível de detalhamento necessário para o gerenciamento de requisitos. Durante o estágio de gerenciamento de requisitos, você deve decidir sobre:

1. *Identificação de requisitos*. Cada requisito deve ser identificado unicamente de modo que possa ser feita a referência cruzada entre este e outros requisitos para que ele possa ser usado nas avaliações de rastreabilidade.
2. *Processo de gerenciamento de mudanças*. É o conjunto de atividades que avaliam o impacto e custo das mudanças. Explico esse processo mais detalhadamente na próxima seção.
3. *Políticas de rastreabilidade*. Essas políticas definem os relacionamentos entre os requisitos e entre os requisitos e o projeto do sistema, que devem ser registrados, e como esses registros devem ser mantidos.
4. *Apoio de ferramentas CASE*. O gerenciamento de requisitos envolve o processamento de grandes quantidades de informações sobre os requisitos. As ferramentas que podem ser usadas variam desde sistemas especializados de gerenciamento de requisitos a planilhas e sistemas simples de banco de dados.

Existem vários relacionamentos entre os requisitos e entre os requisitos e o projeto do sistema. Existem também ligações entre requisitos e os motivos básicos de por que esses requisitos foram propostos. Quando as mudanças são propostas, você deve rastrear seu impacto em outros requisitos e no projeto do sistema. A rastreabilidade é a propriedade de uma especificação de requisitos que reflete a facilidade de encontrar os requisitos relacionados.

Existem três tipos de informações de rastreabilidade que podem ser mantidos:

Tabela 7.1 Classificação de requisitos voláteis

Tipo de requisito	Descrição
Requisitos mutáveis	Requisitos que mudam devido a mudanças no ambiente no qual a organização está operando. Por exemplo, em sistemas hospitalares, o financiamento do tratamento de pacientes pode mudar e, assim, exigir que informações de diferentes tratamentos sejam coletadas.
Requisitos emergentes	Requisitos que surgem à medida que a compreensão do sistema pelo cliente progride durante o desenvolvimento do sistema. O processo de projeto pode revelar novos requisitos emergentes.
Requisitos conseqüentes	Requisitos que resultam da introdução do sistema de computador. A introdução do sistema de computador pode mudar os processos da organização e criar novas formas de trabalho que geram novos requisitos de sistema.
Requisitos de compatibilidade	Requisitos que dependem de sistemas ou processos de negócios específicos dentro de uma organização. À medida que eles mudam, os requisitos de compatibilidade do sistema encomendado ou entregue podem também evoluir.

1. Informações de *rastreabilidade da origem* ligam os requisitos aos stakeholders que propuseram os requisitos e aos motivos desses requisitos. Quando uma mudança é proposta, você usa essas informações para encontrar e consultar os stakeholders sobre a mudança.
2. Informações de *rastreabilidade de requisitos* ligam os requisitos dependentes dentro do documento de requisitos. Você usa essas informações para avaliar quantos requisitos provavelmente serão afetados pela mudança proposta e a extensão das mudanças de requisitos conseqüentes que podem ser necessários.
3. Informações de *rastreabilidade de projeto* ligam os requisitos aos módulos de projeto, nos quais esses requisitos são implementados. Você usa essas informações para avaliar o impacto das mudanças de requisitos propostas no projeto e na implementação do sistema.

As informações de rastreabilidade são freqüentemente representadas por meio de matrizes de rastreabilidade que relacionam os requisitos aos stakeholders, aos outros requisitos ou aos módulos de projeto. Em uma matriz de rastreabilidade de requisitos, cada requisito é introduzido em uma linha e uma coluna da matriz. As dependências entre diferentes requisitos são registradas na célula correspondente à intersecção de linha/coluna.

A Tabela 7.2 mostra uma matriz simples de rastreabilidade que registra as dependências entre requisitos. A letra 'D' na intersecção linha/coluna ilustra que o requisito da linha depende do requisito identificado na coluna; a letra 'R' significa que existe algum outro relacionamento mais fraco entre os requisitos. Por exemplo, ambos podem definir os requisitos para partes do mesmo subsistema.

As matrizes de rastreabilidade podem ser usadas quando um pequeno número de requisitos deve ser gerenciado, mas para sistemas de grande porte, com muitos requisitos, tornam-se muito difíceis de serem gerenciadas e sua manutenção é dispendiosa. Para esses sistemas, você deve captar as informações de rastreabilidade em um banco de dados de requisitos, no qual cada requisito é explicitamente ligado a requisitos relacionados. Você pode então avaliar o impacto das mudanças usando os recursos de acesso ao banco de dados. As matrizes de rastreabilidade podem ser geradas automaticamente baseados no banco de dados.

O gerenciamento de requisitos precisa de apoio automatizado; as ferramentas CASE para essa finalidade devem ser selecionadas durante a fase de planejamento. O apoio de ferramentas é necessário para:

1. *Armazenamento de requisitos*. Os requisitos devem ser mantidos em um repositório de dados seguro e gerenciado acessível a todos os envolvidos no processo de engenharia de requisitos.
2. *Gerenciamento de mudanças*. O processo de gerenciamento de mudanças (Figura 7.10) é simplificado se um apoio ativo de ferramentas estiver disponível.
3. *Gerenciamento de rastreabilidade*. Conforme explicado anteriormente, o apoio de ferramentas para rastreabilidade permite que requisitos relacionados sejam obtidos. Algumas ferramentas usam técnicas de processamento de linguagem natural para auxiliar na descoberta de possíveis relacionamentos entre os requisitos.

Para sistemas de pequeno porte, o uso de ferramentas especializadas de gerenciamento de requisitos pode não ser necessário. O processo de gerenciamento de requisitos pode ter o apoio de recursos disponíveis em processadores de texto, planilhas e bancos de dados de PC. Contudo, para sistemas de grande porte, é necessária uma ferramenta de apoio mais especializada. Incluí links para informações sobre ferramentas de gerenciamento de requisitos, como DOORS e RequisitePro, nas páginas Web do livro.

Tabela 7.2 Matriz de rastreabilidade

ID de requisito	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			R		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

Figura 7.10
Gerenciamento de mudanças de requisitos.

7.4.3 Gerenciamento de mudanças de requisitos

O gerenciamento de mudanças de requisitos (Figura 7.10) deve ser aplicado a todas as mudanças propostas aos requisitos. A vantagem de usar um processo formal para gerenciamento de mudanças é que todas as propostas de mudança são tratadas consistentemente, e as mudanças no documento de requisitos são feitas de maneira controlada. Existem três principais estágios para um processo de gerenciamento de mudanças:

1. *Análise do problema e especificação da mudança*. O processo se inicia com um problema de requisitos identificado ou, às vezes, com uma proposta de mudança específica. Durante esse estágio, o problema ou a proposta de mudança é analisada para verificar se é válida. Os resultados da análise são realimentados para o solicitante da mudança e, algumas vezes, é feita uma proposta de mudança de requisitos mais específica.
2. *Análise da mudança e estimativa de custo*. O efeito da mudança proposta é avaliado usando as informações de rastreabilidade e o conhecimento geral sobre os requisitos do sistema. O custo de realizar a mudança é estimado em termos de modificações no documento de requisitos e, se adequado, do projeto e da implementação do sistema. Uma vez completada esta análise, é tomada uma decisão sobre prosseguir ou não com a mudança de requisitos.
3. *Implementação da mudança*. O documento de requisitos e, quando necessário, o projeto e a implementação de sistema são modificados. Você deve organizar o documento de requisitos de modo que possa realizar as mudanças sem reescrita ou reorganização extensivas. Da mesma maneira que na programação, a facilidade de mudanças no documento é obtida por meio da minimização das referências externas e tornando as seções do documento o mais modulares possível. Assim, as seções individuais podem ser mudadas e substituídas sem afetar outras partes do documento.

Se uma mudança de requisitos do sistema é solicitada com urgência, existe sempre uma propensão a realizar a mudança no sistema e, depois, retrospectivamente, modificar o documento de requisitos. Isso leva quase inevitavelmente à defasagem entre o documento de requisitos e a implementação do sistema. Uma vez feitas as mudanças no sistema, as mudanças no documento de requisitos podem ser esquecidas ou realizadas de maneira não consistente com as mudanças no sistema.

Os processos iterativos de desenvolvimento, como *extreme programming*, foram definidos para lidar com requisitos que mudam durante o processo de desenvolvimento. Nesses processos, quando um usuário propõe mudança de requisito, essa mudança não é realizada por meio de um processo de gerenciamento de mudanças formal. Em vez disso, o usuário precisa priorizar essa mudança e, se a prioridade for alta, decidir quais características do sistema planejadas para a próxima iteração devem ser abandonadas.

PONTOS-CHAVE

- O processo de engenharia de requisitos inclui um estudo de viabilidade, elicitação e análise, especificação, validação e gerenciamento de requisitos.
- A elicitação e a análise de requisitos constituem um processo iterativo que pode ser representado como uma espiral de atividades — obtenção, classificação e organização, negociação e documentação de requisitos.
- Diferentes stakeholders no sistema têm diferentes requisitos. Todos os sistemas complexos devem, portanto, ser analisados com base em uma série de pontos de vista. Pontos de vista podem ser pessoas ou outros sistemas que interagem com o sistema que está sendo especificado, stakeholders afetados pelo sistema ou pontos de vista de domínio que restringem os requisitos.
- Fatores sociais e organizacionais têm forte influência nos requisitos do sistema e podem determinar se o sistema é realmente usado.
- Validação de requisitos é o processo para verificar os requisitos em relação à validade, consistência, completeza, realismo e facilidade de verificação. Revisões de requisitos e prototipação são as principais técnicas usadas para a validação de requisitos.
- Mudanças de negócios, organizacionais e técnicas levam, inevitavelmente, às mudanças dos requisitos de um sistema de software. O gerenciamento de requisitos é o processo para gerenciar e controlar essas mudanças.
- O processo de gerenciamento de requisitos inclui o planejamento de gerenciamento, no qual políticas e procedimentos de gerenciamento de requisitos são projetados, e o gerenciamento de mudanças, no qual você deve analisar as mudanças de requisitos propostas e avaliar seu impacto.

LEITURAS SUGERIDAS

"Requirements engineering". Esta edição especial inclui dois artigos que enfocam a engenharia de software para domínios específicos (automóveis e dispositivos médicos), os quais oferecem perspectivas interessantes sobre os processos de RE nessas áreas. (*IEEE software*, 20 (1), jan./fev. 2003.)

Mastering the requirements process. Um livro de fácil leitura destinado a engenheiros de requisitos com alguma experiência. Fornece orientações específicas para desenvolver um processo eficiente de engenharia de requisitos. (S. Robertson e J. Robertson, 1999, Addison-Wesley.)

Requirements engineering: processes and techniques. Este livro inclui uma visão mais detalhada das atividades do processo de engenharia de requisitos e explica o método VORD e sua aplicação. (G. Kotonya e I. Sommerville, 1999, John Wiley & Sons.)

EXERCÍCIOS

7.1 Sugira quem podem ser os stakeholders em um sistema de registro de estudantes de uma universidade. Explique por que é quase inevitável que os requisitos de diferentes stakeholders sejam conflitantes de alguma forma.

7.2 Um sistema de software será desenvolvido para gerenciar os registros de pacientes internados em uma clínica para tratamento. Os registros incluem todos os registros comuns de monitoração do paciente (temperatura, pressão sangüínea etc.), tratamentos realizados, reações do paciente etc. Após o tratamento, os registros da internação são enviados ao médico do paciente, que mantém seu registro médico completo. Identifique os principais pontos de vista que podem ser levados em consideração na especificação desse sistema e organize-os usando um diagrama de hierarquia de pontos de vista.

7.3 Para três dos pontos de vista identificados no sistema de biblioteca, LIBSYS (Figura 7.4), identifique três requisitos que podem ser sugeridos pelos stakeholders associados a esses pontos de vista.

7.4 O sistema LIBSYS precisa incluir apoio à catalogação de novos documentos, no qual o catálogo do sistema pode ser distribuído em diversas máquinas. Quais são, provavelmente, os tipos de requisitos não funcionais mais importantes associados aos recursos de catalogação?

7.5 Usando seu conhecimento de como um caixa eletrônico é usado, desenvolva um conjunto de casos de uso que pode servir como base para compreensão dos requisitos de um sistema de caixa eletrônico.

7.6 Explique um exemplo de um tipo de sistema no qual os fatores sociais e políticos podem influenciar fortemente os requisitos do sistema. Explique por que esses fatores são importantes em seu exemplo.

7.7 Quem deve ser envolvido em uma revisão de requisitos? Elabore um modelo de processo que mostre como uma revisão de requisitos pode ser organizada.

7.8 Por que as matrizes de rastreabilidade se tornam difíceis de gerenciar quando existem muitos requisitos de sistema? Projete um mecanismo de estruturação de requisitos, baseado em pontos de vista, que possa ajudar a reduzir esse problema.

7.9 Quando mudanças emergenciais precisam ser feitas nos sistemas, o software do sistema pode precisar ser modificado antes da aprovação das mudanças nos requisitos. Sugira um modelo de processo para executar essas modificações que assegure que o documento de requisitos e a implementação do sistema não se tornarão inconsistentes.

7.10 Sua empresa usa um método de análise padrão normalmente aplicado a todas as análises de requisitos. Em seu trabalho, você considera que esse método pode não representar os fatores sociais e organizacionais significativos para o sistema que está sendo analisado. Você expõe isso a seu gerente, que deixa claro que o padrão deve ser seguido. Explique o que você deve fazer em tal situação.

8

Modelos de sistema

Objetivos

O objetivo deste capítulo é apresentar uma série de modelos de sistema que podem ser desenvolvidos durante o processo de engenharia de requisitos. Após ler este capítulo, você:

- compreenderá por que é importante estabelecer os limites de um sistema e modelar seu contexto;
- aprenderá os conceitos da modelagem de comportamento, modelagem de dados e modelagem de objetos;
- compreenderá algumas das notações definidas na Unified Modeling Language (UML) e como essas notações podem ser usadas para desenvolver modelos de sistema.

Conteúdo

8.1 Modelos de contexto
8.2 Modelos de comportamento
8.3 Modelos de dados
8.4 Modelos de objetos
8.5 Métodos estruturados

Os requisitos de usuário devem ser escritos em linguagem natural, pois eles devem ser compreendidos por pessoas que não sejam especialistas técnicos. Contudo, os requisitos mais detalhados do sistema podem ser expressos de maneira mais técnica. Uma técnica amplamente usada é documentar a especificação do sistema como um conjunto de modelos. Esses modelos são representações gráficas que descrevem processos de negócios, o problema a ser resolvido e o sistema a ser desenvolvido. Devido às representações gráficas usadas, os modelos geralmente são mais compreensíveis do que descrições detalhadas dos requisitos de sistema em linguagem natural. Eles também constituem uma importante ponte entre os processos de análise e de projeto.

Você pode usar modelos no processo de análise para obter uma compreensão maior do sistema existente a ser substituído ou aprimorado, ou especificar o novo sistema necessário. Você pode desenvolver diferentes modelos para representar o sistema de diferentes perspectivas. Por exemplo:

1. Uma perspectiva externa, na qual o contexto ou o ambiente do sistema é modelado.
2. Uma perspectiva comportamental, na qual o comportamento do sistema é modelado.
3. Uma perspectiva estrutural, na qual a arquitetura do sistema ou a estrutura dos dados processados pelo sistema é modelada.

Abordo essas três perspectivas neste capítulo e explico também a modelagem de objetos, que combina, em certa extensão, as modelagens comportamental e estrutural.

O aspecto mais importante de um modelo de sistema é que ele deixa os detalhes de fora. Um modelo de sistema é uma abstração em estudo, e não uma representação alternativa do sistema. De modo ideal, uma *representação* do sistema deve manter toda as informações sobre a entidade que está sendo representada. Uma *abstração* simplifica intencionalmente a descrição e ressalta as características mais salientes. Por exemplo, no improvável evento de este livro ser publicado em um jornal, a apresentação, no caso, seria uma abstração dos principais pontos do livro. Se ele fosse traduzido do inglês para o italiano, o livro traduzido seria uma representação alternativa. A intenção do tradutor seria manter todas as informações conforme apresentadas em inglês.

Diferentes tipos de modelos de sistema são baseados em diferentes abordagens para abstração. Um modelo de fluxo de dados, por exemplo, concentra-se no fluxo de dados e nas transformações funcionais desses dados. Ele deixa de fora os detalhes das estruturas de dados. Por outro lado, um modelo de entidades de dados e seus relacionamentos documenta as estruturas de dados do sistema em vez de sua funcionalidade.

Exemplos dos tipos de modelos de sistema que podem ser criados durante o processo de análise são:

1. *Um modelo de fluxo de dados*. Modelos de fluxo de dados mostram como os dados são processados em diferentes estágios do sistema.
2. *Um modelo de composição*. Um modelo de composição ou *agregação* mostra como as entidades no sistema são compostas por outras entidades.
3. *Um modelo de arquitetura*. Modelos de arquitetura mostram os principais subsistemas que constituem um sistema.
4. *Um modelo de classificação*. Diagramas de classe de objeto/herança mostram como as entidades têm características comuns.
5. *Um modelo de estímulo-resposta*. Um modelo de estímulo-resposta, ou *diagrama de transição de estados*, mostra como o sistema reage aos eventos internos e externos.

Todos esses tipos de modelos são abordados neste capítulo. Sempre que possível, uso notações da Unified Modeling Language (UML), que se tornou uma linguagem padrão para modelagem orientada a objetos (Booch, et al., 1999; Rumbaugh, et al., 1999a). Quando a UML não possui notações apropriadas, uso notações simples e intuitivas para a descrição do modelo. Uma nova versão da UML (UML 2.0) está em desenvolvimento, mas não estava disponível quando este capítulo foi escrito. No entanto, entendo que a notação UML que uso neste capítulo seja, provavelmente, compatível com a UML 2.0.

8.1 Modelos de contexto

Em um estágio inicial do processo de elicitação e análise de requisitos, você deve decidir sobre os limites do sistema. Isso implica trabalhar com os stakeholders no sistema para distinguir o que é sistema e o que é o ambiente do sistema. Você deve tomar essas decisões no início do processo para limitar os custos do sistema e o tempo necessário para análise.

Em alguns casos, o limite entre um sistema e seu ambiente é relativamente claro. Por exemplo, quando um sistema automatizado substitui um sistema manual ou baseado em computador, o ambiente do novo sistema é geralmente o mesmo do sistema existente. Em outros casos, existe mais flexibilidade e você pode decidir o que constitui o limite entre o sistema e seu ambiente durante o processo de engenharia de requisitos.

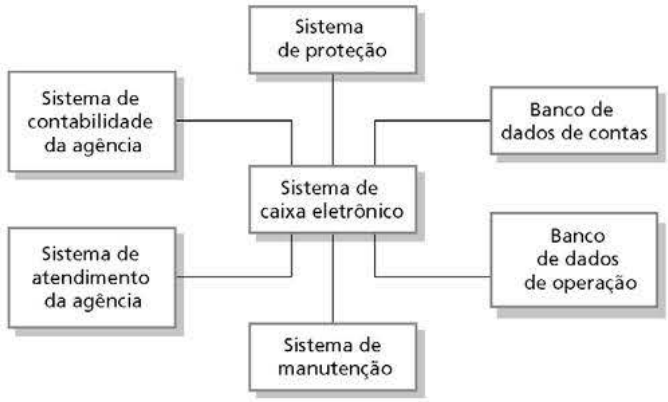
Por exemplo, digamos que você esteja desenvolvendo a especificação do sistema LIBSYS de biblioteca. Lembre-se de que esse sistema destina-se a fornecer versões eletrônicas de material com direitos autorais para os computadores dos usuários. Os usuários podem, assim, imprimir cópias pessoais do material. Ao desenvolver a especificação desse sistema, você precisa decidir se outros sistemas de banco de dados de biblioteca, como catálogos de biblioteca, estão dentro dos limites do sistema. Se estiverem, você precisa permitir o acesso ao sistema por meio da interface com o usuário de catálogo; se não estiverem, os usuários podem sofrer o incômodo de ter de mudar de um sistema para outro.

A definição de um limite de sistema não é um julgamento livre de valores. As considerações sociais e organizacionais podem significar que a posição de um limite de sistema pode ser determinada por fatores não técnicos. Por exemplo, o limite de um sistema pode ser definido de modo que o processo de análise possa ser realizado totalmente em um local; ele pode ser escolhido de modo que um gerente particularmente antipático não precise ser consultado; pode ser definido de modo que o custo do sistema aumente e a área de desenvolvimento de sistemas deva, portanto, expandir-se para projetar e implementar o sistema.

Após a tomada de algumas decisões sobre os limites do sistema, parte da atividade de análise é a definição desse contexto e as dependências que um sistema tem com seu ambiente. Normalmente, a produção de um modelo de arquitetura simples é o primeiro passo dessa atividade.

Figura 8.1

Contexto de um sistema de caixa eletrônico.



A Figura 8.1 é um modelo de arquitetura que ilustra a estrutura do sistema de informações que inclui uma rede de caixas eletrônicos bancários. Os modelos de arquitetura de alto nível são geralmente expressos como diagramas de blocos simples, nos quais cada subsistema é representado por um retângulo identificado e as linhas indicam associações entre os subsistemas.

Da Figura 8.1, observamos que cada caixa eletrônico está conectado a um banco de dados de contas, um sistema de contabilidade da agência, um sistema de proteção e um sistema para apoiar a manutenção da máquina. O sistema está também conectado a um banco de dados de operação, que monitora como a rede de caixas eletrônicos é usada, e ao sistema de caixas da agência local. Esse sistema de balcão fornece serviços, como back-up e impressão. Portanto, esses serviços não precisam ser incluídos no próprio sistema de caixa eletrônico.

Os modelos de arquitetura descrevem o ambiente de um sistema. No entanto, eles não mostram os relacionamentos entre os outros sistemas no ambiente e o sistema que está sendo especificado. Os sistemas externos podem produzir dados para o sistema ou receber dados gerados pelo sistema. Eles podem compartilhar dados com o sistema ou estar diretamente conectados, por meio de uma rede, ou nenhum desses casos. Eles podem estar fisicamente próximos ou localizados em prédios separados. Todas essas relações podem afetar os requisitos do sistema que está sendo definido e devem ser levadas em consideração.

Portanto, os modelos de arquitetura simples são normalmente complementados com outros modelos, como modelos de processo que mostram as atividades de processo apoiadas pelo sistema. Os modelos de fluxo de dados (descritos na seção seguinte) podem também ser usados para mostrar os dados transferidos entre o sistema e outros sistemas em seu ambiente.

A Figura 8.2 ilustra um modelo de processo de aquisição de equipamentos em uma organização. Ele consiste em especificar o equipamento necessário, procurar e escolher fornecedores, elaborar o pedido do equipamento, receber o equipamento e testá-lo após a entrega. Ao especificar o apoio computacional para esse processo, você deve decidir quais dessas atividades serão realmente apoiadas. As outras atividades estão além dos limites do sistema. Na Figura 8.2, a linha tracejada agrupa as atividades dentro dos limites do sistema.

8.2 Modelos de comportamento

Os modelos de comportamento são usados para descrever o comportamento geral do sistema. Explico dois tipos de modelos de comportamento nesta seção: modelos de fluxo de dados, que modelam o processamento de dados no sistema, e modelos de máquina de estado, que modelam como o sistema reage aos eventos. Esses modelos podem ser usados separadamente ou em conjunto, dependendo do tipo do sistema que está sendo desenvolvido.

A maioria dos sistemas de negócios é dirigida principalmente por dados. É controlada pelas entradas de dados no sistema com, relativamente, pouco processamento de eventos externos. O modelo de fluxo de dados pode ser tudo de que se precisa para representar o comportamento desses sistemas. De modo contrário, sistemas de tempo real são freqüentemente dirigidos por eventos, com um mínimo de processamento de dados. Um modelo de máquina de estado (explicado na Seção 8.2.2) é a maneira mais eficiente de representar seu comportamento. Outras classes de sistemas podem ser dirigidas tanto por dados como por eventos. Nesses casos, você pode desenvolver os dois tipos de modelo.

8.2.1 Modelos de fluxo de dados

Os modelos de fluxos de dados constituem uma maneira intuitiva de mostrar como os dados são processados por um sistema. No nível de análise, eles devem ser usados para modelar a maneira com que os dados são processados no

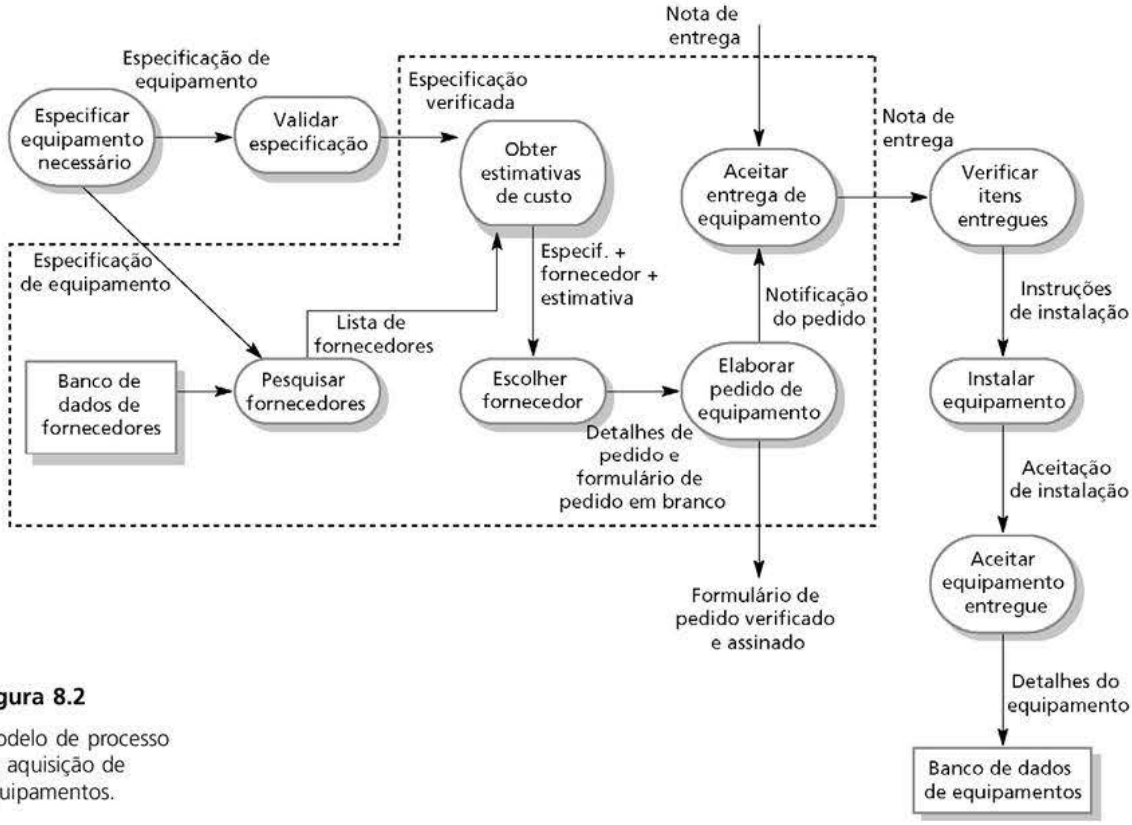


Figura 8.2

Modelo de processo de aquisição de equipamentos.

sistema existente. O uso de modelos de fluxo de dados para análise tornou-se comum após a publicação do livro de DeMarco (DeMarco, 1978) sobre análise estruturada de sistemas. Os modelos de fluxo de dados constituem uma parte intrínseca dos métodos estruturados desenvolvidos com base nesse trabalho. A notação usada nesses modelos representa o processamento funcional (retângulos arredondados), repositórios de dados (retângulos) e movimentos de dados entre funções (setas).

Os modelos de fluxo de dados são usados para mostrar como os dados fluem por uma seqüência de etapas de processamento. Por exemplo, uma etapa de processamento pode ser filtrar os registros duplicados em um banco de dados de cliente. Os dados são transformados a cada etapa, antes de passarem ao próximo estágio. Essas etapas de processamento ou transformações representam os processos ou funções de software quando os diagramas de fluxo de dados são usados para documentar um projeto de software. Contudo, em um modelo de análise, as pessoas ou os computadores podem realizar o processamento.

Um modelo de fluxo de dados, que mostra as etapas envolvidas no processamento de um pedido de produtos (como equipamentos de computador) em uma organização, é ilustrado na Figura 8.3. Esse modelo específico descreve o processamento de dados na atividade **Elaborar pedido de equipamentos** no modelo geral de processo mostrado na Figura 8.2. O modelo mostra como o pedido de produtos passa de processo a processo. Também são mostrados os repositórios de dados (**Arquivo de pedidos** e **Arquivo de orçamento**) envolvidos nesse processo.

Os modelos de fluxo de dados são valiosos porque acompanham e documentam como os dados associados a determinado processo se movem pelo sistema. Eles ajudam os analistas a compreender o que está acontecendo. Os diagramas de fluxo de dados têm a vantagem de serem simples e intuitivos, diferentemente de outras notações de modelagem. Em geral é possível explicá-los para os potenciais usuários do sistema, que podem então participar da validação da análise.

Em princípio, o desenvolvimento de modelos, como os modelos de fluxo de dados, deve ser um processo top-down (de cima para baixo). Nesse exemplo, isso implica começar pela análise do processo geral de aquisição. Em seguida, você deve passar à análise de subprocessos, como o de realizar o pedido. Na prática, a análise nunca ocorre dessa maneira. Você aprende sobre os vários níveis ao mesmo tempo. Os modelos de níveis mais baixos podem ser desenvolvidos primeiro e, depois, são abstraídos para criar um modelo mais geral.

Os modelos de fluxo de dados mostram uma perspectiva funcional, na qual cada transformação representa uma única função ou processo. Eles são particularmente úteis durante a análise de requisitos, pois podem ser usados para mostrar o

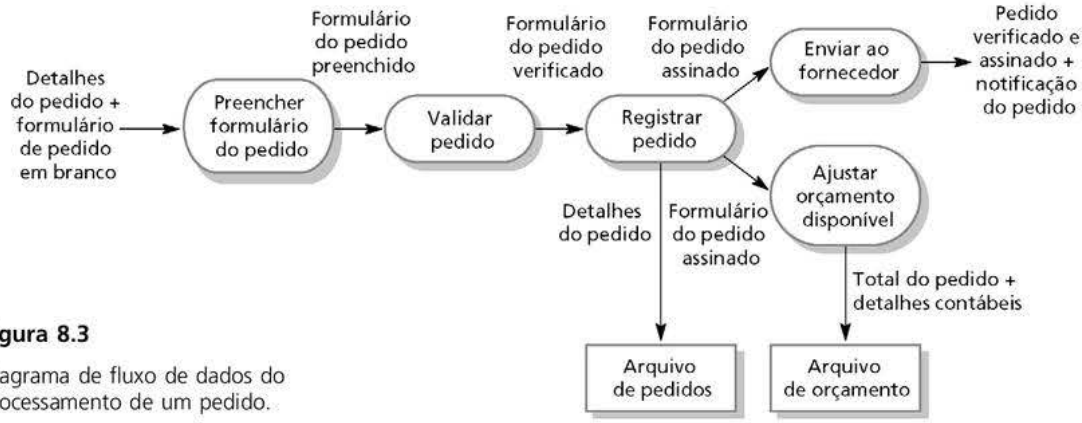


Figura 8.3

Diagrama de fluxo de dados do processamento de um pedido.

processamento de ponta a ponta em um sistema. Ou seja, eles mostram a seqüência completa das ações que ocorrem a partir de uma entrada, que é processada, até a correspondente saída, que é a resposta do sistema. A Figura 8.4 ilustra esse uso de diagramas de fluxo de dados. É um diagrama de processamento que ocorre no sistema de bomba de insulina, apresentado no Capítulo 3.

8.2.2 Modelos de máquina de estado

Um modelo de máquina de estado descreve como um sistema responde aos eventos internos ou externos. O modelo de máquina de estado mostra os estados do sistema e os eventos que provocam transições de um estado para outro. Ele não mostra o fluxo de dados dentro do sistema. Esse tipo de modelo é freqüentemente usado para modelagem de sistemas de tempo real, porque estes sistemas geralmente são dirigidos pelos estímulos do ambiente do sistema. Por exemplo, o sistema de alarme de tempo real, explicado no Capítulo 13, responde aos estímulos de sensores de movimento, sensores de abertura de portas etc.

Os modelos de máquina de estado são parte integrante de métodos de projeto de tempo real, como os propostos por Ward e Mellor (Ward e Mellor, 1985) e Harel (Harel, 1987; Harel, 1988). O método de Harel usa a notação chamada statecharts que foram a base para a notação de modelagem de máquina de estado na UML.

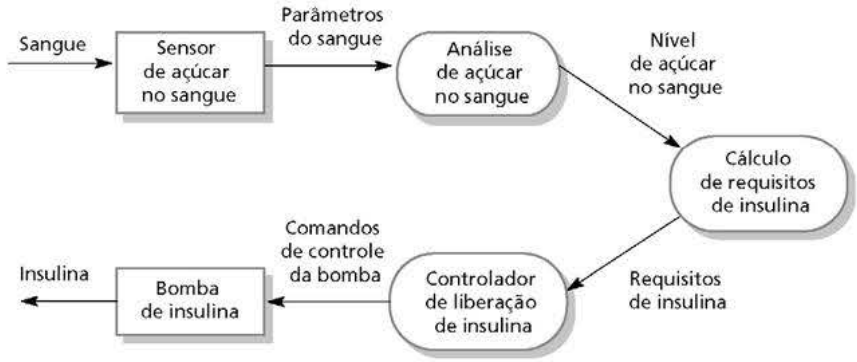
Um modelo de máquina de estado supõe que, a qualquer instante, o sistema está em um dos vários estados possíveis. Quando um estímulo é recebido, ele pode disparar uma transição para um estado diferente. Por exemplo, um sistema de controle de válvula pode passar do estado 'válvula aberta' para o estado 'válvula fechada' quando é recebido um comando (estímulo) do operador.

Essa abordagem de modelagem de sistema está ilustrada na Figura 8.5. Esse diagrama mostra um modelo de máquina de estado de um forno de microondas simples equipado com botões para definir a potência e o relógio e para iniciar o sistema. Os fornos de microondas reais são muito mais complexos que o sistema descrito nesse exemplo. Entretanto, esse modelo inclui as principais características do sistema. Para simplificar o modelo, supus que a seqüência de ações no uso do forno de microonda é:

1. Selecionar o nível de potência (meia potência ou potência total).
2. Inserir o tempo de cozimento.
3. Pressionar Iniciar e o alimento será cozido durante o tempo definido.

Figura 8.4

Diagrama de fluxo de dados de uma bomba de insulina.



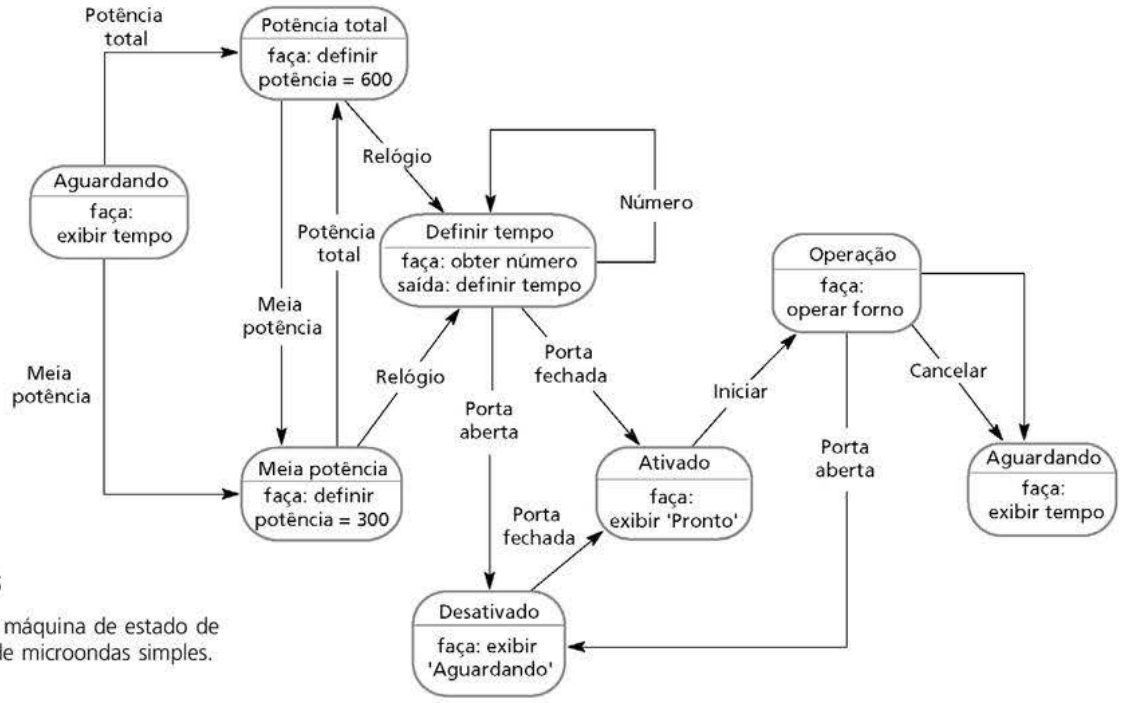


Figura 8.5

Modelo de máquina de estado de um forno de microondas simples.

Por razões de segurança, o forno não deve funcionar quando a porta estiver aberta e, ao término do cozimento, um alarme é acionado. O forno tem um display alfanumérico muito simples, usado para exibir várias mensagens de alerta e de aviso.

A notação UML, que uso para descrever os modelos de máquina de estado, é projetada para modelar o comportamento de objetos. Contudo, é uma notação de propósito geral que pode ser usada para qualquer tipo de modelagem de máquina de estado. Os retângulos de cantos arredondados no modelo representam os estados do sistema. Eles incluem uma breve descrição (depois de 'faça') das ações executadas nesse estado. As setas representam os estímulos que forçam a transição de um estado para outro.

Portanto, na Figura 8.5, podemos observar que o sistema responde inicialmente tanto para o botão de potência total como para o botão de meia potência. Os usuários podem mudar de idéia após a seleção de um desses botões e pressionar o outro botão. O tempo é definido e, se a porta estiver fechada, o botão Iniciar é ativado. Quando esse botão é pressionado, inicia-se a operação do forno e o cozimento é realizado durante o tempo especificado.

A notação UML permite que você indique a atividade que ocorre em um estado. Em uma especificação de sistema detalhada, você deve fornecer mais detalhes sobre os estímulos e os estados do sistema (Tabela 8.1). Essa informação pode ser mantida em um dicionário de dados ou uma enciclopédia (abordada na Seção 8.3) mantida pelas ferramentas CASE usadas para criar o modelo do sistema.

O problema com a abordagem de máquinas de estado é que o número de possíveis estados cresce rapidamente. Para modelos de sistemas de grande porte, portanto, é necessária alguma estruturação desses modelos de estados. Uma maneira de fazer isso é usar a noção de um superestado que engloba uma série de estados separados. Esse superestado parece um único estado no modelo de alto nível, mas é, então, expandido em mais detalhes em um diagrama separado. Para ilustrar esse conceito, considere o estado **Operação** na Figura 8.5. Esse é um superestado que pode ser expandido, conforme ilustrado na Figura 8.6.

O estado **Operação** inclui uma série de subestados. A Figura 8.6 mostra que a operação se inicia com uma verificação do status e, se algum problema for detectado, um alarme será acionado e a operação, desativada. O cozimento envolve ativar o gerador de microondas por determinado tempo; no término desse tempo, um alarme sonoro será acionado. Se a porta for aberta durante a operação, o sistema passará ao estado desativado, conforme mostrado na Figura 8.5.

8.3 Modelos de dados

A maioria dos sistemas de software faz uso de um grande banco de dados de informações. Em alguns casos, esse banco de dados é independente do sistema de software. Em outros, ele é criado para o sistema que está sendo desen-

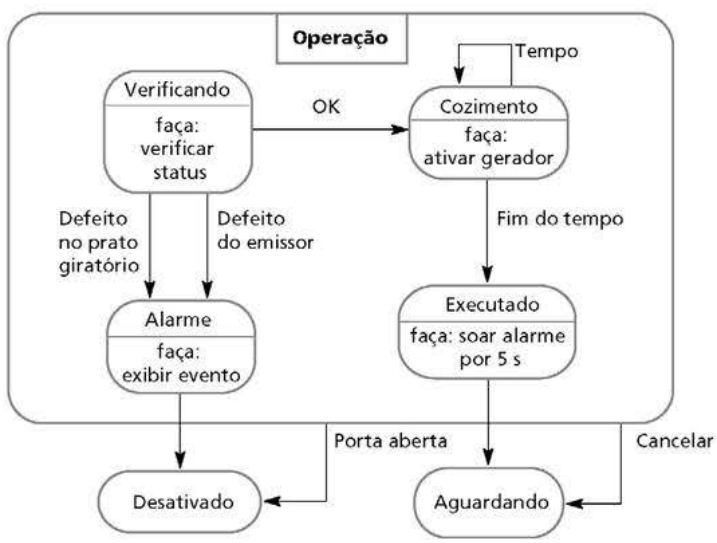
Tabela 8.1 Descrição de estados e estímulos do forno de microondas

Estado	Descrição
Aguardando	O forno está aguardando uma entrada. O display apresenta a hora atual.
Meia potência	A potência do forno é definida como 300 watts. O display apresenta 'Meia potência'.
Potência total	A potência do forno é definida como 600 watts. O display apresenta 'Potência total'.
Definir tempo	O tempo de cozimento é definido como o valor fornecido pelo usuário. O display apresenta o tempo de cozimento selecionado e é atualizado quando o tempo for definido.
Desativado	A operação do forno é desativada por questões de segurança. A luz interna do forno é ligada. O display apresenta 'Não está pronto'.
Ativado	A operação do forno é ativada. A luz interna do forno é apagada. O display apresenta 'Pronto para cozinhar'.
Operação	Forno em operação. A luz interna do forno é ligada. O display apresenta a contagem regressiva do relógio. Ao término do cozimento, o alarme soa por 5 segundos. A luz do forno é ligada. O display apresenta 'Cozimento concluído' enquanto o alarme soa.

Estímulo	Descrição
Meia potência	O usuário pressionou o botão de meia potência.
Potência total	O usuário pressionou o botão de potência total.
Relógio	O usuário pressionou um dos botões do relógio.
Número	O usuário pressionou uma tecla numérica.
Porta aberta	A trava da porta do forno não está fechada.
Porta fechada	A trava da porta do forno está fechada.
Iniciar	O usuário pressionou o botão Iniciar.
Cancelar	O usuário pressionou o botão Cancelar.

Figura 8.6

Operação do forno de microondas.



volvido. Uma parte importante da modelagem de sistemas é a definição da forma lógica dos dados processados pelo sistema. Eles são algumas vezes chamados de *modelos semânticos de dados*.

A técnica de modelagem de dados mais amplamente usada é a modelagem Entidade-Relacionamento-Atributo (ERA — Entity-Relation-Attribute), que mostra as entidades de dados, seus atributos associados e as relações entre essas enti-

dades. Essa abordagem de modelagem foi proposta inicialmente em meados da década de 1970 por Chen (Chen, 1976); muitas variações foram desenvolvidas desde então (Codd, 1979; Hammer e McLeod, 1981; Hull e King, 1987), todas com a mesma forma básica.

Os modelos de entidade-relacionamento foram amplamente usados no projeto de banco de dados. Os esquemas de bancos de dados relacionais derivados desses modelos estão naturalmente em sua terceira forma normal, que é uma característica desejável (Barker, 1989). Por causa dos tipos explícitos e do reconhecimento de subtipos e supertipos, a implementação desses modelos usando os bancos de dados orientados a objetos é também direta.

A UML não inclui uma notação específica para essa modelagem de banco de dados, uma vez que assume um processo de desenvolvimento orientado a objetos e modela os dados usando objetos e seus relacionamentos. Contudo, você pode usar a UML para representar um modelo semântico de dados. Você pode pensar em entidades de um modelo ERA como classes simplificadas de objetos (elas não possuem operações), em atributos como atributos de classe e nas associações identificadas entre as classes como relações.

A Figura 8.7 é um exemplo de modelo de dados que é parte do sistema LIBSYS de biblioteca apresentado nos capítulos iniciais. Lembre-se de que o LIBSYS é projetado para fornecer cópias de artigos com direitos autorais, publicados em revistas, e para receber os pagamentos por esses artigos. Portanto, o modelo de dados deve incluir informações sobre o artigo, o detentor dos direitos autorais e o comprador do artigo. Presumi que os pagamentos pelos artigos não são feitos diretamente aos detentores, mas sim por agências nacionais de direitos autorais.

A Figura 8.7 mostra que um **Artigo** tem atributos que representam o título, os autores, o nome do arquivo PDF do artigo e a taxa a ser paga. Ele é ligado à **Fonte** na qual o artigo foi publicado e à **Agência de Direitos Autorais** do país de publicação. Tanto a **Agência de Direitos Autorais** quanto a **Fonte** estão ligados a **País**. O país da publicação é importante, pois as leis de direitos autorais variam de acordo com o país. O diagrama mostra também que o **Comprador** faz o **Pedido** referente a **Artigos**.

Como todos os modelos gráficos, os modelos de dados são pobres em detalhes e você deve manter descrições mais detalhadas das entidades, relacionamentos e atributos incluídos no modelo. Você pode obter essas descrições mais detalhadas em um repositório ou dicionário de dados. Os dicionários de dados são geralmente úteis quando são desenvolvidos modelos de sistema e podem ser usados para gerenciar todas as informações de todos os tipos de modelos de sistema.

Um dicionário de dados é, de modo simplificado, uma lista alfabética dos nomes incluídos nos modelos de sistema. Além do nome, o dicionário de dados inclui uma descrição associada à entidade com nome e, se o nome representar um objeto composto, inclui uma descrição da composição. Outras informações, como a data de criação, o autor e a representação da entidade podem também ser incluídas, dependendo do tipo de modelo que está sendo desenvolvido.

Figura 8.7

Modelo semântico de dados para o sistema LIBSYS.

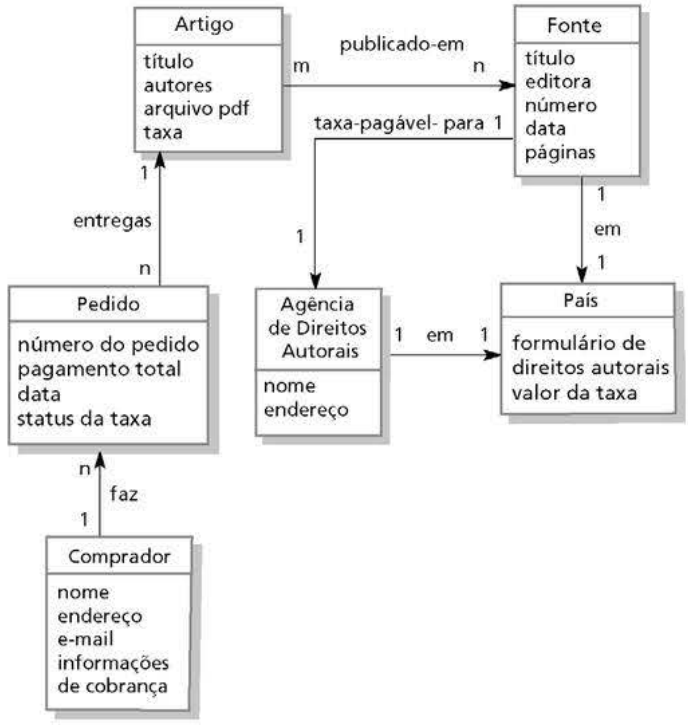


Tabela 8.2 Exemplos de entradas do dicionário de dados

Nome	Descrição	Tipo	Data
Artigo	Detalhes do artigo publicado que pode ser pedido pelas pessoas que usam o LIBSYS.	Entidade	30.12.2002
Autores	Nomes dos autores do artigo que podem receber uma parte da taxa.	Atributo	30.12.2002
Comprador	A pessoa ou organização que pede uma cópia do artigo.	Entidade	30.12.2002
Taxa-pagável-para	Relacionamento 1:1 entre Artigo e Agência de Direitos Autorais à qual deve ser paga a taxa de direitos autorais.	Relação	29.12.2002
Endereço (comprador)	Endereço do comprador. É usado para quaisquer informações de cobrança necessárias.	Atributo	31.12.2002

As vantagens do uso de um dicionário de dados são:

1. *É um mecanismo para gerenciamento de nomes.* Muitas pessoas podem precisar inventar nomes para entidades e relacionamentos ao desenvolver um modelo de sistema de grande porte. Esses nomes devem ser usados consistentemente e não devem entrar em conflito. O software de dicionário de dados pode verificar a exclusividade do nome, quando for necessário, e avisar aos analistas sobre requisitos de duplicidade de nomes.
2. *Serve como repositório de informações organizacionais.* À medida que o sistema for sendo desenvolvido, as informações associadas à análise, ao projeto, à implementação e à evolução são acrescentadas no dicionário de dados, de modo que toda informação sobre uma entidade esteja armazenada em um único lugar.

As entradas de dicionário de dados, apresentadas na Tabela 8.2, definem os nomes no modelo semântico de dados para o LIBSYS (Figura 8.7). Simplifiquei a apresentação desse exemplo excluindo alguns nomes e abreviando as informações associadas.

Todos os nomes de sistema, sejam eles de entidades, relações, atributos ou serviços, devem ser inseridos no dicionário de dados. Normalmente usa-se software para criar, manter e consultar o dicionário. Esse software pode estar integrado a outras ferramentas, de modo que a criação do dicionário seja parcialmente automatizada. Por exemplo, as ferramentas CASE que apóiam a modelagem de sistemas geralmente incluem apoio aos dicionários de dados e inserem os nomes no dicionário quando são usados pela primeira vez no modelo.

8.4 Modelos de objetos

Uma abordagem orientada a objetos para todo o processo de desenvolvimento de software é atualmente de uso comum, particularmente no desenvolvimento de sistemas interativos. Isso significa expressar os requisitos de sistemas usando um modelo de objetos, projetar usando objetos e desenvolver o sistema por meio de uma linguagem de programação orientada a objetos, como Java ou C++.

Os modelos de objetos desenvolvidos durante a análise de requisitos podem ser usados para representar tanto os dados do sistema quanto seu processamento. Nesse sentido, eles combinam alguns modelos de fluxo de dados e modelos semânticos de dados. São também úteis para demonstrar como as entidades do sistema podem ser classificadas e compostas por outras entidades.

Para algumas classes de sistemas, os modelos de objetos são maneiras naturais de refletir as entidades do mundo real, manipuladas pelo sistema. Isso é particularmente verdadeiro, quando o sistema processa informações sobre entidades tangíveis, como automóveis, aeronaves ou livros, os quais possuem atributos claramente identificáveis. As entidades mais abstratas, de nível mais alto, como conceitos de uma biblioteca, um sistema de registros médicos ou um processador de texto, são mais difíceis de serem modeladas como classes de objetos. Elas não têm necessariamente uma interface simples, constituída de atributos e operações independentes.

O desenvolvimento de modelos de objetos durante a análise de requisitos em geral simplifica a transição para o projeto e a programação orientada a objetos. Entretanto, descobri que os usuários finais de um sistema geralmente consideram os modelos de objetos não naturais e difíceis de serem compreendidos. Eles podem preferir uma visão mais funcional de processamento de dados. Portanto, algumas vezes, os modelos de fluxo de dados ajudam a complementar os modelos de objetos, que mostram o processamento de dados de ponta a ponta em um sistema.

Uma classe de objeto é uma abstração de um conjunto de objetos que identifica atributos comuns (como em um modelo semântico de dados) e os serviços ou operações fornecidos por cada objeto. Os objetos são entidades executáveis com os atributos e serviços de uma classe de objeto. Os objetos são instâncias da classe de objeto e vários objetos podem ser criados baseados em uma classe. Normalmente, os modelos desenvolvidos usando a análise enfocam as classes de objeto e os relacionamentos entre elas.

Na análise de requisitos orientada a objetos, você deve modelar as entidades do mundo real usando as classes de objeto. Você não deve incluir detalhes dos objetos individuais (instâncias da classe) no sistema. Deve criar diferentes tipos de modelos de objeto mostrando como as classes de objeto estão relacionadas umas com as outras, como os objetos são agregados para formar outros objetos, como os objetos interagem com outros objetos etc. Cada um desses modelos apresenta uma única informação sobre o sistema que está sendo especificado.

O processo de análise para identificar os objetos e as classes de objeto é considerado uma das áreas mais difíceis do desenvolvimento orientado a objetos. A identificação de objetos é basicamente a mesma para análise e projeto. Os métodos de identificação de objetos, abordados no Capítulo 14, que explica o projeto orientado a objetos, podem ser usados. Nesta seção concentro-me em alguns dos modelos de objetos que podem ser gerados durante o processo de análise.

Vários métodos de análise orientada a objetos foram propostos na década de 1990 (Coad e Yourdon, 1990; Rumbaugh, et al., 1991; Jacobsen, et al., 1993; Booch, 1994). Esses métodos tinham muitas coisas em comum e três dos principais desenvolvedores (Booch, Rumbaugh e Jacobsen) decidiram integrar suas abordagens para produzir um método unificado (Rumbaugh, et al., 1999b). A linguagem de modelagem unificada (UML — Unified Modeling Language) usada nesse método unificado tornou-se um padrão para a modelagem de objetos. A UML inclui notações para diferentes tipos de modelos de sistema. Já vimos os modelos de casos de uso e os diagramas de seqüência nos capítulos iniciais, além dos modelos de máquina de estado no início deste capítulo.

Uma classe de objeto em UML, conforme ilustrado nos exemplos da Figura 8.8, é representada como um retângulo vertical com três seções:

1. O nome da classe de objeto está na seção superior.
2. Os atributos de classe estão na seção intermediária.
3. As operações associadas à classe de objeto estão na seção inferior do retângulo.

Não há espaço para abordar toda a UML, portanto, concentro-me nos modelos de objetos que mostram como os objetos podem ser classificados e herdar atributos e operações de outros objetos, modelos de agregação que mostram como os objetos são compostos e modelos de comportamento simples, que mostram as interações entre os objetos.

8.4.1 Modelos de herança

A modelagem orientada a objetos envolve a identificação de classes de objeto importantes no domínio em estudo. Essas classes são, portanto, organizadas em uma taxonomia. Uma taxonomia é um esquema de classificação que mostra como uma classe de objeto está relacionada a outras classes por meio de atributos e serviços comuns.

Para apresentar essa taxonomia, as classes são organizadas em uma hierarquia de herança, com as classes de objeto mais genéricas no topo da hierarquia. Os objetos mais especializados herdam seus atributos e serviços. Esses objetos especializados podem ter seus próprios atributos e serviços.

A Figura 8.8 ilustra parte de uma hierarquia simplificada de classes para um modelo de biblioteca. Essa hierarquia fornece informações sobre os itens mantidos na biblioteca. A biblioteca mantém vários itens, como livros, música, registros de filmes, revistas e jornais. Na Figura 8.8, o item mais genérico está no topo da árvore e tem um conjunto de atributos e serviços que são comuns a todos os itens da biblioteca. Eles são herdados pelas classes **Item publicado** e **Item registrado**, que acrescentam seus próprios atributos que são, então, herdados pelos itens de nível inferior.

A Figura 8.9 é um exemplo de outra hierarquia de herança que pode ser parte do modelo de biblioteca. Nesse caso, são mostrados os usuários de uma biblioteca. Existem duas classes de usuários: aqueles que podem solicitar o empréstimo de livros e aqueles que podem apenas ler os livros na biblioteca, sem retirá-los.

Na notação UML, a herança é mostrada no 'sentido ascendente' ao invés de no 'sentido descendente', como acontece em outras notações orientadas a objetos ou em linguagens, como Java, em que as subclasses herdam de superclasses. Isto é, a seta (mostrada como um triângulo) aponta a partir das classes, que herdam atributos e operações, para a superclasse. Em vez de usar o termo *herança*, a UML se refere ao *relacionamento de generalização*.

O projeto de hierarquias de classes não é fácil e, portanto, o analista precisa compreender, detalhadamente, o domínio no qual o sistema será instalado. Como exemplo da sutileza dos problemas que surgem na prática, considere a hierarquia de itens da biblioteca. Aparentemente, o atributo **Título** poderia ser colocado no item geral e ser herdado por itens de nível inferior.

No entanto, embora todos os itens de uma biblioteca devam ter algum tipo de identificador ou número de registro, isso não indica que tudo deva ter um título. Por exemplo, uma biblioteca pode manter material pessoal de um político aposentado.

Figura 8.8

Parte da hierarquia de classes de uma biblioteca.

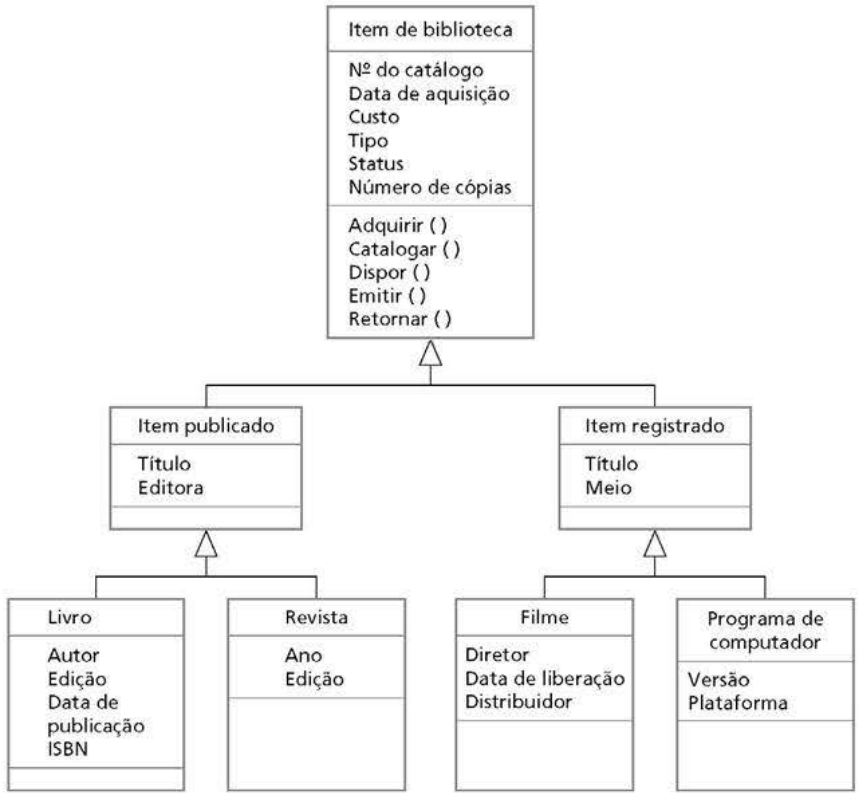
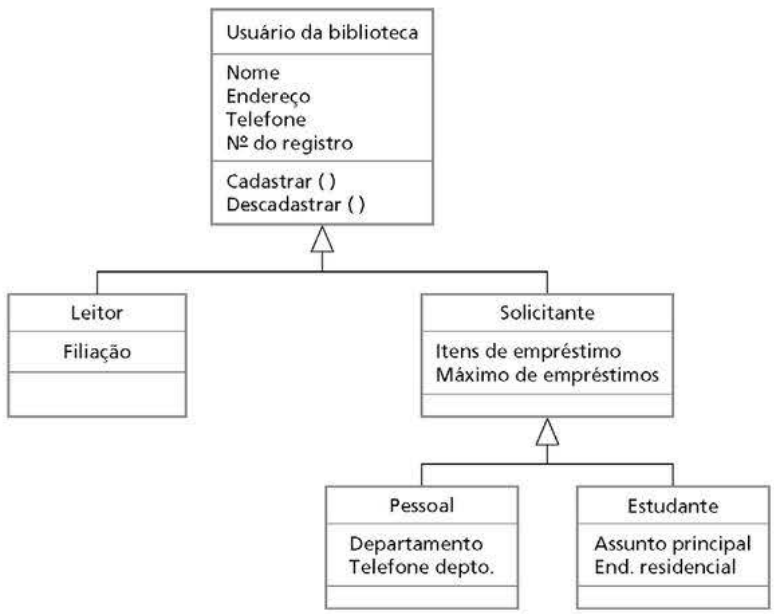


Figura 8.9

Hierarquia de classes de usuário.



Muitos desses itens, como cartas, podem não ser explicitamente identificados. Eles serão classificados usando alguma outra classe (não mostrada no exemplo) que tenha um conjunto diferente de atributos.

As figuras 8.8 e 8.9 mostram as hierarquias de herança de classes, nas quais toda classe de objeto herda seus atributos e operações de uma única classe-pai. Modelos de herança múltipla nos quais uma classe tem vários pais também podem ser construídos. Seus atributos e serviços herdados constituem um conjunto daqueles herdados de cada superclasse. A Figura 8.10 mostra um exemplo de modelo de herança múltipla que pode também ser parte do modelo de biblioteca.

O principal problema com a herança múltipla é projetar um gráfico de herança no qual os objetos não herdem atributos desnecessários. Outros problemas incluem a dificuldade da reorganização do gráfico de herança quando são necessárias

Figura 8.10

Herança múltipla.

mudanças e da resolução de conflitos de nomes em que os atributos de duas ou mais superclasses tenham o mesmo nome, mas com diferentes significados. No nível de modelagem de sistema, tais conflitos são relativamente fáceis de resolver, alterando-se manualmente o modelo de objetos. Eles causam problemas maiores na programação orientada a objetos.

8.4.2 Agregação de objetos

Assim como atributos e serviços são herdados por meio de um relacionamento de herança com outros objetos, alguns objetos são agrupamentos de outros objetos. Isto é, um objeto é uma *agregação* de um conjunto de outros objetos. As classes que representam esses objetos podem ser modeladas por meio de um modelo de agregação de objetos, conforme mostrado na Figura 8.11. Neste exemplo, foi modelado um item de biblioteca que é um pacote de estudo para um curso universitário. Esse pacote de estudo inclui anotações de aulas, exercícios, exemplos de soluções, cópias de slides usados nas aulas e fitas de vídeo.

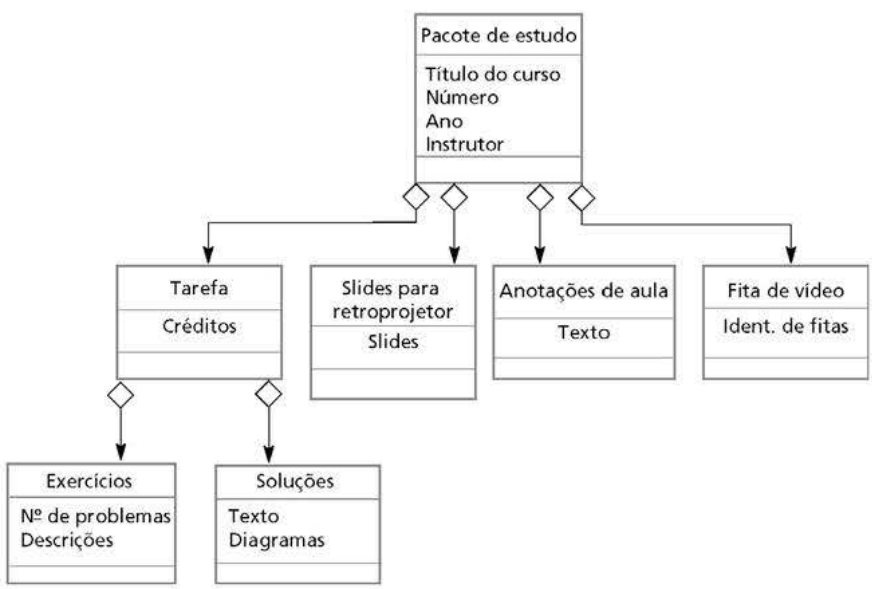
A notação UML para agregação se destina a representar a composição pela inclusão de um losango na origem da ligação. Portanto, a Figura 8.11 pode ser lida da seguinte forma: 'Um pacote de estudo é composto por uma ou mais tarefas, slides para retroprojetor, anotações de aulas e fitas de vídeo'.

8.4.3 Modelagem de comportamento de objetos

Para modelar o comportamento de objetos, você precisa mostrar como as operações fornecidas pelos objetos são usadas. Em UML, os comportamentos são modelados usando cenários representados como casos de uso de UML (explicados no Capítulo 7). Uma maneira de modelar o comportamento é usar diagramas de seqüência que mostram a seqüência das ações envolvidas em um caso de uso. Assim como os diagramas de seqüência, a UML também inclui diagramas de

Figura 8.11

Objeto agregado que representa um curso.



colaboração que mostram a seqüência de mensagens trocadas pelos objetos. Eles são similares aos diagramas de seqüência; sendo assim, não os abordo nesta seção.

Você pode observar como os diagramas de seqüência podem ser usados para modelagem de comportamento na Figura 8.12, que expande o caso de uso do sistema LIBSYS, no qual os usuários retiram itens da biblioteca em formato eletrônico. Por exemplo, imagine a situação em que os pacotes de estudo, mostrados na Figura 8.11, podem ser mantidos eletronicamente e baixados no computador do estudante.

Em um diagrama de seqüência, os objetos e os agentes estão alinhados na parte superior do diagrama. As setas rotuladas indicam as operações; a seqüência das operações ocorre de cima para baixo. Nesse cenário, o usuário da biblioteca acessa o catálogo para verificar se o item desejado está disponível eletronicamente; se estiver, o usuário solicitará a cópia eletrônica desse item. Por motivos de direitos autorais, a cópia deve ser licenciada e, por isso, existe uma transação entre o item e o usuário, na qual ocorre a aceitação da licença. O item a ser copiado é, então, enviado a um objeto servidor de rede para ser comprimido antes de ser enviado ao usuário da biblioteca.

Você pode encontrar outro exemplo de um diagrama de seqüência, que expande um caso de uso do sistema LIBSYS, na Figura 7.7, o qual mostra a seqüência de ações envolvida na impressão de um artigo.

8.5 Métodos estruturados

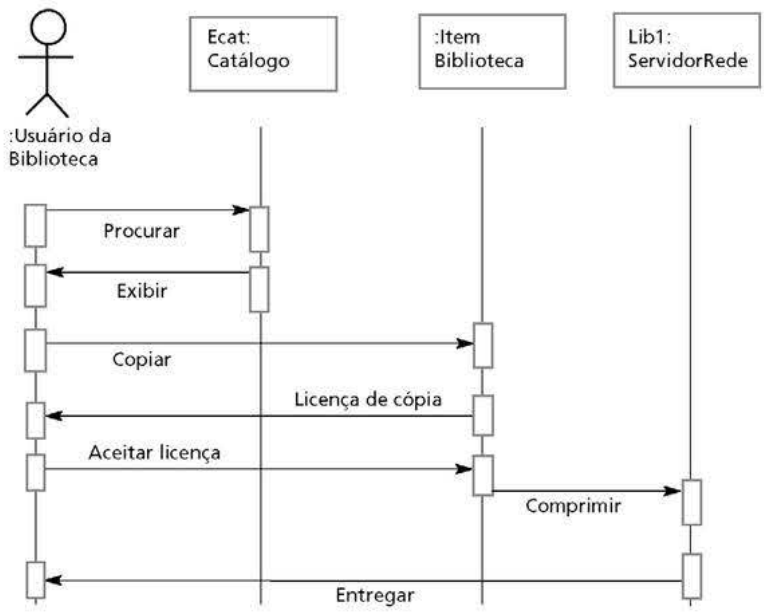
Um método estruturado é uma maneira sistemática de produção de modelos de um sistema existente ou de um sistema a ser construído. Esses métodos foram inicialmente desenvolvidos na década dos 1970 para apoiar a análise e o projeto de software (Constantine e Yourdon, 1979; Gane e Sarson, 1979; Jackson, 1983) e evoluiu nas décadas de 1980 e 1990 para apoiar o desenvolvimento orientado a objetos (Rumbaugh, et al., 1991; Robinson, 1992; Jacobsen, et al., 1993; Booch, 1994). Esses métodos orientados a objetos se fundiram: a UML proposta como uma linguagem padrão de modelagem (Booch, et al., 1999; Rumbaugh, et al., 1999a) e o Unified Process (Rumbaugh, et al., 1999b); mais tarde, o Rational Unified Process (Krutchen, 2000), como um método estruturado associado. Budgen (Budgen, 2003) resume e compara vários desses métodos estruturados.

Os métodos estruturados fornecem um framework para modelagem detalhada do sistema, como parte da elicitação e análise de requisitos. A maioria dos métodos estruturados tem seu próprio conjunto preferencial de modelos de sistema. Eles geralmente definem um processo que pode ser usado para derivar esses modelos e um conjunto de regras e diretrizes que se aplicam a esses modelos. Uma documentação padronizada é produzida para o sistema. As ferramentas CASE em geral estão disponíveis para apoiar o método. Essas ferramentas apóiam a edição de modelos e geração de códigos e relatórios, e fornecem alguns recursos de verificação de modelos.

Os métodos estruturados foram aplicados, com sucesso, em vários projetos de grande porte. Eles podem levar a significativas reduções de custo, pois usam notações padronizadas e asseguram que uma documentação padronizada do projeto seja produzida. Contudo, os métodos estruturados apresentam uma série de pontos fracos:

Figura 8.12

Cópia de itens eletrônicos.



1. Não fornecem apoio eficiente para compreender ou modelar os requisitos não funcionais do sistema.
2. São indiscriminados, pois geralmente não incluem diretrizes para ajudar os usuários a decidirem se um método é apropriado para determinado problema. Também não incluem, normalmente, orientações sobre como eles podem ser adaptados para uso em um ambiente específico.
3. Freqüentemente produzem um excesso de documentação. A essência dos requisitos de sistema pode ser ocultada pela quantidade de detalhes incluídos.
4. Os modelos produzidos são muito detalhados, e os usuários em geral os consideram difíceis de serem compreendidos. Esses usuários, portanto, não podem verificar o aspecto real desses modelos.

Na prática, no entanto, os engenheiros de requisitos e os projetistas não se restringem aos modelos propostos por determinado método. Por exemplo, os métodos orientados a objetos não sugerem geralmente que os modelos de fluxos de dados devem ser desenvolvidos. Contudo, em minha experiência, esses modelos normalmente são úteis como parte de um processo de análise de requisitos, pois podem apresentar uma visão geral do processamento do sistema de ponta a ponta. Eles podem também contribuir diretamente com a identificação de objetos (os dados que fluem) e a identificação de operações sobre esses objetos (as transformações).

As ferramentas CASE para análise e projeto apóiam a criação, edição e análise das notações gráficas usadas nos métodos estruturados. A Figura 8.13 mostra os componentes que podem ser incluídos no ambiente de apoio ao método.

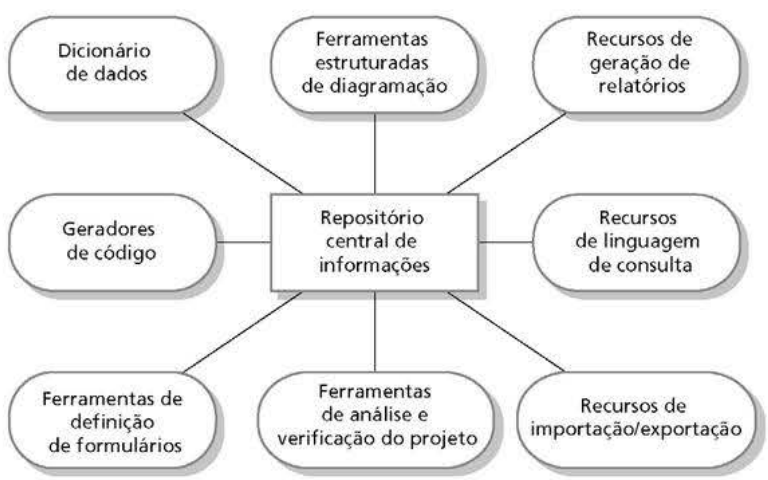
As ferramentas abrangentes de apoio ao método, conforme ilustradas na Figura 8.13, normalmente incluem:

1. *Editores de diagramas* usados para criar modelos de objeto, modelos de dados, modelos de comportamento etc. Esses editores não são apenas ferramentas para desenhar, mas têm consciência dos tipos de entidades no diagrama. Eles captam as informações sobre essas entidades e gravam as informações no repositório central.
2. *Ferramentas de análise e verificação de projeto* que processam o projeto e relatam os erros e as anormalidades. Elas podem ser integradas com o sistema de edição, de modo que os erros dos usuários sejam detectados no estágio inicial do processo.
3. *Linguagens de consulta ao repositório* que permitem ao projetista encontrar projetos e informações associadas no repositório.
4. *Um dicionário de dados* que mantém informações sobre as entidades usadas em um projeto de sistema.
5. *Ferramentas de definição e geração de relatórios* que obtêm informações do repositório central e geram automaticamente a documentação do sistema.
6. *Ferramentas de definição de formulários* que permitem que formatos de telas e documentos sejam especificados.
7. *Recursos de importação/exportação* que permitem o intercâmbio de informações do repositório central com outras ferramentas de desenvolvimento.
8. *Geradores de código* que geram códigos ou esqueletos de código automaticamente com base no projeto captado no repositório central.

Os conjuntos de ferramentas CASE mais abrangentes permitem que o usuário gere um programa ou um fragmento de programa baseado em informações existentes no modelo do sistema. As ferramentas CASE freqüentemente apóiam linguagens diferentes, de modo que o usuário pode gerar um programa em C, C++ ou Java com base no mesmo modelo de projeto. Como

Figura 8.13

Componentes de uma ferramenta CASE para apoio de método estruturado



os modelos excluem detalhes de nível inferior, o gerador de código em um workbench de projeto geralmente não pode criar o sistema completo. Freqüentemente é necessário codificar manualmente para adicionar os detalhes ao código gerado.

PONTOS-CHAVE

- Um modelo é uma visão abstrata de um sistema que ignora alguns detalhes de sistema. Modelos complementares do sistema podem ser desenvolvidos para apresentar outras informações sobre o sistema.
- Os modelos de contexto mostram como o sistema modelado está posicionado em um ambiente com outros sistemas e processos. Eles definem os limites do sistema. Modelos de arquitetura, modelos de processo e modelos de fluxo de dados podem ser usados como modelos de contexto.
- Os diagramas de fluxo de dados podem ser usados para modelar o processamento de dados realizado por um sistema. O sistema é modelado como um conjunto de transformações de dados, sendo que as funções atuam sobre os dados.
- Os modelos de máquina de estado são usados para modelar o comportamento do sistema em resposta a eventos internos ou externos.
- Os modelos semânticos de dados descrevem a estrutura lógica dos dados importados e exportados pelo sistema. Esses modelos mostram as entidades do sistema, seus atributos e os relacionamentos dos quais participam.
- Os modelos de objetos descrevem as entidades lógicas do sistema e sua classificação e agregação. Eles combinam um modelo de dados com um modelo de processamento. Os modelos de objetos possíveis que podem ser desenvolvidos incluem modelos de herança, modelos de agregação e modelos de comportamento.
- Os modelos de seqüência, que mostram as interações entre agentes e objetos em um sistema, são usados para modelar o comportamento dinâmico.
- Os métodos estruturados fornecem um framework para apoiar o desenvolvimento de modelos de sistema. Eles normalmente têm um apoio extenso de ferramentas CASE, incluindo edição e verificação de modelos, e geração de código.

LEITURAS SUGERIDAS

Software design, 2nd ed. Embora este livro seja principalmente voltado ao projeto de software, o autor explica uma série de métodos estruturados que podem também ser usados no processo de engenharia de requisitos. Ele não enfoca apenas abordagens orientadas a objetos. (D. Budgen, 2003, Addison-Wesley.)

Requirements analysis and system design. Este livro enfoca a análise de sistemas de informações e explica como modelos diferentes de UML podem ser usados no processo de análise. (L. Maciaszek, 2001, Addison-Wesley.)

Software engineering with objects and components. Uma introdução breve e de fácil leitura ao uso da UML na especificação e projeto de sistemas. Embora seja menos abrangente que a descrição completa da UML, este livro será muito mais adequado caso você esteja tentando aprender e compreender a notação. (P. Stevens com R. Pooley, 1999, Addison-Wesley.)

EXERCÍCIOS

8.1 Elabore um modelo de contexto para um sistema de informações de pacientes em um hospital. Você pode formular quaisquer hipóteses razoáveis sobre outros sistemas hospitalares disponíveis, mas seu modelo deve incluir um sistema de admissão de pacientes e um sistema para armazenamento de imagens de raios X, bem como outros registros de diagnóstico.

8.2 Baseado em sua experiência com um caixa eletrônico bancário, elabore um diagrama de fluxo de dados modelando o processamento de dados envolvido quando um cliente retira dinheiro da máquina.

8.3 Modele o processamento de dados que pode ocorrer em um sistema de e-mail. Você deve modelar separadamente o processamento de envio e de recebimento de mensagens.

8.4 Elabore modelos de máquina de estado do software de controle para:

- Uma máquina de lavar automática com diferentes programas para diferentes tipos de roupas
- O software de um equipamento de DVD
- Um sistema de secretária eletrônica de telefone que registra as mensagens recebidas e apresenta o número de mensagens aceitas por meio de um LED. O sistema deve permitir que o usuário do telefone ligue de qualquer local, digite uma seqüência de números (identificados como tons) e ouça as mensagens registradas.

8.5 Um modelo de sistema de software pode ser representado como um gráfico direcionado, no qual os nós são as entidades do modelo e os arcos são os relacionamentos entre as entidades. As entidades e os relacionamentos no modelo podem ser rotulados com um nome e outras informações. Cada entidade no modelo tem um tipo e pode ser 'explodida' em um submodelo. Elabore um modelo de dados que descreva a estrutura de um modelo de sistema de software.

8.6 Modele as classes de objeto que podem ser usadas em um sistema de e-mail. Se você tentou fazer o Exercício 8.3, descreva as similaridades e diferenças entre o modelo de processamento de dados e o modelo de objetos.

8.7 Usando as informações sobre os dados de sistema, mostradas na Figura 8.7, elabore um diagrama de seqüência que mostre uma possível seqüência de ações que ocorrem quando um novo artigo é catalogado pelo sistema LIBSYS.

8.8 Desenvolva um modelo de objetos incluindo um diagrama de hierarquia de classes e um diagrama de agregação que mostre os principais componentes de um sistema de computador pessoal e seu software de sistema.

8.9 Desenvolva um diagrama de seqüência que mostre as interações envolvidas quando um estudante se matricula em um curso de uma universidade. Os cursos podem ter vagas limitadas, de modo que o processo de matrícula deve incluir verificações de se as vagas estão disponíveis. Suponha que o estudante acessa um catálogo eletrônico de cursos para conhecer os cursos disponíveis.

8.10 Em que circunstâncias você não recomendaria o uso de métodos estruturados para o desenvolvimento de sistemas?

9

Especificação de sistemas críticos

Objetivos

O objetivo deste capítulo é explicar como especificar os requisitos funcionais e não funcionais de confiabilidade para sistemas críticos. Após ler este capítulo, você:

- compreenderá como os requisitos de confiabilidade podem ser identificados para sistemas críticos pela análise dos riscos enfrentados por esses sistemas;
- verá que os requisitos de segurança são gerados com base na análise de riscos do sistema e não com base nos stakeholders do sistema;
- conhecerá o processo de derivação dos requisitos de proteção e como esses requisitos são gerados para combater diferentes tipos de ameaças ao sistema;
- compreenderá as métricas para a especificação de confiabilidade e como essas métricas podem ser usadas para especificar os requisitos de confiabilidade.

Conteúdo

9.1 Especificação dirigida a riscos
9.2 Especificação de segurança
9.3 Especificação de proteção
9.4 Especificação de confiabilidade de software

Em setembro de 1993, um avião aterrissou no aeroporto de Varsóvia, na Polônia, durante uma tempestade com relâmpagos e trovões. Por nove segundos após a aterrissagem, os freios do sistema de freios controlado por computador não funcionaram. O avião ultrapassou o fim da pista, colidiu contra um banco de terra e pegou fogo. A investigação posterior mostrou que o software do sistema de freios funcionou perfeitamente de acordo com sua especificação. No entanto, por motivos que não explicarei neste capítulo, o sistema de freios não reconheceu que o avião tinha aterrissado. Uma característica de segurança da aeronave interrompeu o funcionamento do sistema de freios, pois essa situação poderia ser perigosa se o avião estivesse no ar. A falha no sistema foi causada devido a um erro na especificação do sistema.

Isso ilustra a importância da especificação para sistemas críticos. Devido aos potenciais altos custos de falha no sistema, é importante assegurar que a especificação para sistemas críticos reflita precisamente as necessidades reais dos usuários do sistema. Se você não obtiver a especificação correta, então, independentemente da qualidade do desenvolvimento do software, o sistema não será confiável.

A necessidade de confiabilidade em sistemas críticos gera os requisitos funcionais e não funcionais do sistema:

1. Requisitos funcionais do sistema podem ser gerados para definir os recursos de verificação e de recuperação de erros e características para fornecer proteção contra falhas do sistema.
2. Os requisitos não funcionais podem ser gerados para definir a confiabilidade e a disponibilidade necessárias ao sistema.

Além desses requisitos, as considerações de segurança e proteção podem gerar outro tipo de requisito difícil de ser classificado como um requisito funcional ou não funcional. Eles são requisitos de alto nível mais bem descritos, talvez, como requisitos do tipo 'não deve'. Ao contrário dos requisitos funcionais normais que definem o que o sistema deve fazer, os requisitos do tipo 'não deve' definem um comportamento inaceitável do sistema. Exemplos de requisitos do tipo 'não deve' são:

O sistema não deve permitir que usuários modifiquem permissões de acesso de quaisquer arquivos que eles não tenham criado (proteção).

O sistema não deve permitir a seleção do modo reverso de empuxo quando a aeronave estiver em vôo (segurança).

O sistema não deve permitir a ativação simultânea de mais de três sinais de alarme (segurança).

Esses requisitos do tipo 'não deve' são, algumas vezes, decompostos em requisitos funcionais de software mais específicos. De modo alternativo, as decisões de implementação podem ser adiadas até que o sistema seja projetado.

Os requisitos de usuário de sistemas críticos serão sempre especificados com a utilização de linguagem natural e modelos de sistema. Contudo, conforme explico no Capítulo 10, a especificação formal e a verificação associada são, provavelmente, eficazes em termos de custo no desenvolvimento de sistemas críticos (Hall, 1996; Hall e Chapman, 2002; Wordsworth, 1996). As especificações formais não são apenas uma base para a verificação do projeto e da implementação. Elas constituem a maneira mais precisa de especificação de sistemas e, portanto, reduzem as chances de mal-entendidos. Além disso, a criação de uma especificação formal força uma análise detalhada dos requisitos, o que é uma maneira eficiente de descobrir problemas na especificação. Em uma especificação em linguagem natural, os erros podem ser ocultados pela imprecisão da linguagem. Isso não ocorre se o sistema for formalmente especificado.

9.1 Especificação dirigida a riscos

A especificação dos sistemas críticos complementa o processo normal de especificação de requisitos, enfocando a confiabilidade do sistema. Seu objetivo é compreender os riscos enfrentados pelo sistema e gerar requisitos de confiabilidade para lidar com esses riscos. A especificação dirigida a riscos foi amplamente usada pelos desenvolvedores de sistemas críticos de segurança e de proteção. Em sistemas críticos de segurança, os riscos são os perigos que podem resultar em acidentes; em sistemas críticos de proteção, os riscos são as vulnerabilidades que podem tornar um ataque bem-sucedido. Devido à crescente importância da proteção, incluí uma explicação detalhada de métodos dirigidos a riscos no Capítulo 31, que aborda a engenharia de proteção, na nova seção de Tecnologias Emergentes.

O processo de especificação dirigido a riscos envolve a compreensão dos riscos enfrentados pelo sistema, a descoberta das causas de origem e a geração de requisitos para gerenciar esses riscos. A Figura 9.1 mostra o processo iterativo de análise de riscos:

1. *Identificação de riscos.* Os riscos potenciais que podem ocorrer são identificados. Eles dependem do ambiente no qual o sistema será usado.
2. *Análise e classificação de riscos.* Os riscos são considerados separadamente. Os potencialmente graves e possíveis são selecionados para análise posterior. Nesse estágio, alguns riscos podem ser eliminados simplesmente por ser muito improvável que ocorram (por exemplo, queda de energia e terremoto simultâneos).

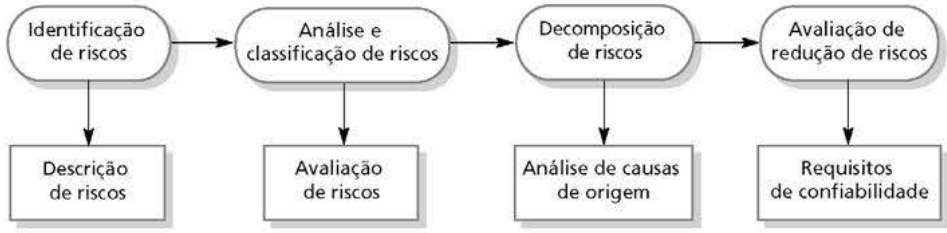


Figura 9.1
Especificação dirigida a riscos.

3. *Decomposição de riscos*. Cada risco é analisado individualmente para descobrir as respectivas causas de origem potenciais. Técnicas como a análise de árvore de defeitos (explicada mais adiante neste capítulo) podem ser usadas.
4. *Avaliação de redução de riscos*. São feitas propostas de como os riscos identificados podem ser reduzidos ou eliminados. Essas propostas geram, portanto, os requisitos de confiabilidade do sistema, que definem as defesas contra o risco e como o risco será gerenciado, caso ocorra.

Para sistemas de grande porte, a análise de riscos pode ser estruturada em fases. A análise de riscos de várias fases é necessária para sistemas de grande porte, como indústrias químicas ou aeronaves. As fases da análise de riscos incluem:

- Análise preliminar de riscos, na qual os riscos principais são identificados
- Análise de riscos mais detalhada do sistema e dos subsistemas
- Análise de riscos de software, na qual os riscos de falha de software são considerados
- Análise de riscos operacionais relacionados à interface com o usuário do sistema e com os riscos que surgem devido a erros de operador.

Leveson (Leveson, 1995) explica esse processo de análise de riscos de várias fases em seu livro sobre sistemas críticos de segurança.

9.1.1 Identificação de riscos

O objetivo da identificação de riscos, que é o primeiro estágio do processo de análise de riscos, é identificar os riscos com os quais o sistema crítico deve lidar. Esse processo pode ser complexo e difícil, pois os riscos freqüentemente ocorrem devido às interações entre o sistema e condições ambientais raras. O acidente de Varsóvia, que expliquei inicialmente, ocorreu quando os ventos transversais à rota da aeronave, devido a uma tempestade, causaram sua inclinação, de modo que o avião aterrissou sobre uma roda em vez de duas.

Nos sistemas críticos de segurança, os principais riscos são os perigos que podem levar a um acidente. Você pode enfrentar o problema de identificação de perigos considerando classes diferentes de perigo, como perigos físicos, elétricos, biológicos, radioativos, de falhas de serviço etc. Cada uma dessas classes pode ser analisada para descobrir os perigos associados. Possíveis combinações de perigos devem também ser identificadas.

Apresentei um exemplo de um sistema de bomba de insulina no Capítulo 3. Como vários dispositivos médicos, ele é um sistema crítico de segurança. Alguns perigos ou riscos que poderiam ocorrer nesse sistema são:

1. Dose excessiva de insulina (falha de serviço)
2. Dose insuficiente de insulina (falha de serviço)
3. Falha de energia devido ao desgaste da bateria (elétrico)
4. Interferência elétrica com outros equipamentos médicos, como marca-passo cardíaco (elétrico)
5. Mau contato de sensor e atuador, causado pelo ajuste incorreto (físico)
6. Partes do equipamento quebradas no corpo do paciente (físico)
7. Infecção causada pela introdução do equipamento (biológico)
8. Reação alérgica aos materiais ou à insulina usados no equipamento (biológico).

Os riscos relativos ao software estão normalmente relacionados à falha de fornecimento de um serviço ou à falha de sistemas de monitoração e proteção. Os sistemas de monitoração podem detectar condições potencialmente perigosas, como falhas de energia.

Os engenheiros experientes, que trabalham junto com especialistas do domínio e consultores profissionais de segurança, devem identificar os riscos do sistema. Técnicas de trabalho em grupo, como *brainstorming*, podem ser usadas para identificar os riscos. Os analistas, com experiência direta obtida com incidentes anteriores, podem também ser capazes de identificar os riscos.

9.1.2 Análise e classificação de riscos

O processo de análise e classificação de riscos liga-se principalmente com a compreensão da probabilidade de ocorrência de um risco e das conseqüências potenciais, se um acidente ou incidente, associado com este risco, ocorrer. Precisamos fazer essa análise para compreender se um risco é uma ameaça séria ao sistema ou ao ambiente, e para fornecer uma base para decidir os recursos que devem ser usados para gerenciar o risco.

Para cada risco, a saída do processo de análise e classificação de riscos é uma declaração de aceitação. Os riscos podem ser categorizados de três maneiras:

1. *Intolerável*. O sistema deve ser projetado de modo que o risco não possa ocorrer ou, se ocorrer, não resulte em um acidente. Riscos intoleráveis seriam, tipicamente, aqueles que ameaçam a vida humana ou a estabilidade financeira de um negócio e que têm uma probabilidade significativa de ocorrência. Um exemplo de risco intolerável para um sistema de e-commerce em uma livraria na Internet, digamos, seria o risco de o sistema ficar fora de funcionamento durante mais de um dia.
2. *Tão baixo quanto razoavelmente prático (*ALARP — *As low as reasonably practical)*. O sistema deve ser projetado de modo que a probabilidade de ocorrência de um acidente, devido ao perigo, seja mínima, sujeita a outras considerações, como custo e entrega. Os riscos ALARP são os que têm conseqüências menos sérias ou que têm baixa probabilidade de ocorrência. Um risco ALARP para um sistema de e-commerce poderia ser o corrompimento de imagens da página Web que apresenta o logotipo da empresa. Isso é comercialmente indesejável, mas é improvável que cause sérias conseqüências em curto prazo.
3. *Aceitável*. Enquanto os projetistas de sistema devem tomar todas as providências possíveis para reduzir a probabilidade da ocorrência de um perigo 'aceitável', essas providências não devem aumentar os custos, o tempo de entrega ou outros atributos não funcionais do sistema. Um exemplo de risco aceitável para um sistema de e-commerce seria aquele em que as pessoas que usam uma versão beta de navegadores Web não possam completar seus pedidos com sucesso.

A Figura 9.2 (Brazendale e Bell, 1994), desenvolvida para sistemas críticos de segurança, mostra essas três regiões. O formato do diagrama reflete os custos para assegurar que os riscos não resultem em incidentes ou acidentes. O custo de projeto do sistema para poder lidar com o risco é uma função da largura do triângulo. Os custos mais altos incorrem devido aos riscos do topo do diagrama e os custos mais baixos devido aos riscos no vértice inferior do triângulo.

Os limites entre as regiões da Figura 9.2 tendem a se mover ao longo do tempo devido às expectativas públicas de segurança e a considerações políticas. Embora os custos financeiros de aceitação dos riscos e de indenização por quaisquer acidentes resultantes possam ser menores do que os custos de prevenção do acidente, a opinião pública pode exigir que os custos adicionais devam ser aceitos. Por exemplo, pode ser mais barato para uma empresa limpar um vazamento nas raras vezes que isso ocorrer em vez de instalar sistemas para prevenção de poluição. Essa medida pode ter sido aceitável nas décadas de 1960 e 1970, mas não é pública e politicamente aceitável nos dias de hoje. O limite entre a região intolerável e a região ALARP moveu-se para baixo, de modo que os riscos que podiam ser aceitos no passado são intoleráveis atualmente.

A avaliação de riscos envolve a estimativa da probabilidade de riscos e da gravidade deles. Isso geralmente é muito difícil de ser feito de maneira exata e, freqüentemente, depende do julgamento da engenharia. As probabilidades e as gravidades são atribuídas usando-se termos relativos, como *provável, improvável* e *rara*, e também *alta, média* e *baixa.* As experiências anteriores com o sistema podem permitir associar algum valor numérico a esses termos. Entretanto, como os acidentes são relativamente incomuns, é muito difícil validar a precisão desse valor.

A Tabela 9.1 mostra uma classificação de riscos (perigos), identificados na seção anterior, para o sistema de liberação de insulina. Como não sou médico, incluí as estimativas apenas para ilustrar o princípio. Elas não são necessariamente as probabilidades e as gravidades reais que ocorreriam em uma análise real de um sistema de fornecimento de insulina. Observe que uma dose excessiva de insulina é potencialmente mais grave do que uma dose insuficiente no curto prazo. A dose excessiva de insulina pode causar disfunções orgânicas, coma e, por fim, morte.

Figura 9.2
Níveis de risco.

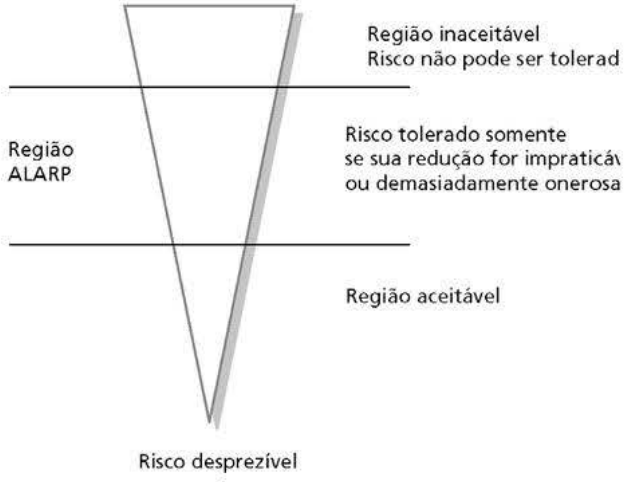


Tabela 9.1 Análise de riscos de perigos identificados em uma bomba de insulina

Perigo identificado	Probabilidade do perigo	Gravidade do perigo	Risco estimado	Nível de aceitação
1. Dose excessiva de insulina	Média	Alta	Alto	Intolerável
2. Dose insuficiente de insulina	Média	Baixa	Baixo	Aceitável
3. Falha de energia	Alta	Baixa	Baixo	Aceitável
4. Equipamento ajustado incorretamente	Alta	Alta	Alto	Intolerável
5. Quebra do equipamento durante o uso no paciente	Baixa	Alta	Médio	ALARP
6. Equipamento causa infecção	Média	Média	Médio	ALARP
7. Interferência elétrica	Baixa	Alta	Médio	ALARP
8. Reação alérgica	Baixa	Baixa	Baixo	Aceitável

Os perigos de 3 a 8 não estão relacionados ao software e, portanto, não serão explicados além desta seção. Para conter esses perigos, o equipamento deve ter software *built-in* de autoteste que deve monitorar o estado do sistema e alertar sobre alguns desses perigos. O alerta permitirá que o perigo seja detectado antes que cause um acidente. Exemplos de perigos que devem ser detectados são a falha de energia e o ajuste incorreto do equipamento. O software de monitoração está, naturalmente, relacionado à segurança no caso de falha em detectar um perigo poder resultar em um acidente.

9.1.3 Decomposição de riscos

A decomposição de riscos é o processo de descobrir as causas da origem dos riscos em determinado sistema. As técnicas de decomposição de riscos foram derivadas principalmente do desenvolvimento de sistemas críticos de segurança, nos quais a análise de perigos é parte fundamental do processo de segurança. A análise de riscos pode ser dedutiva ou indutiva. As técnicas dedutivas (*top-down*) tendem a ser mais fáceis de usar, começam com o risco e trabalham, tomando-o como base, em direção a uma possível falha de sistema; as técnicas indutivas (*bottom-up*) começam com uma falha de sistema proposta e identificam quais perigos ocorreriam e que poderiam levar a essa falha.

Várias técnicas foram propostas como possíveis abordagens para a decomposição de riscos. Elas incluem revisões e checklists, bem como técnicas mais formais, como análise de rede de Petri (Peterson, 1981), lógica formal (Jahanian e Mok, 1986) e análise de árvore de defeitos (Leveson e Stolzy, 1987; Storey, 1996).

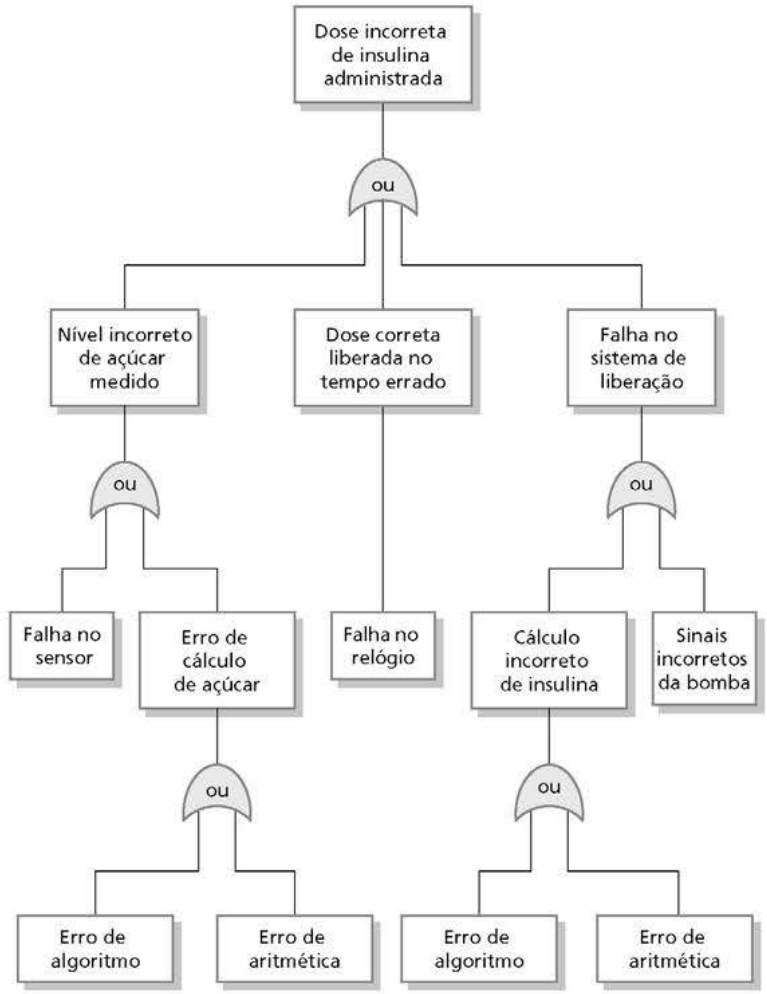
Nesta seção, abordo a análise de árvore de defeitos. Essa técnica foi desenvolvida para sistemas críticos de segurança e é relativamente fácil de compreender sem o conhecimento de domínio de um especialista. A análise de árvore de defeitos envolve a identificação do evento não desejado e a análise retroativa a partir desse evento para descobrir as possíveis causas do perigo. O perigo é colocado na raiz da árvore e são identificados os estados que podem levar a esse perigo. Para cada um desses estados, você deve identificar os estados que podem levar a ele e prossegue com a decomposição até identificar as causas de origem do risco. Os estados podem ser ligados por meio de símbolos 'e' e 'ou'. Os riscos que requerem uma combinação de causas de origem são geralmente menos prováveis do que os riscos que podem resultar de uma única causa de origem.

A Figura 9.3 é a árvore de defeitos para os perigos relacionados com o software do sistema de liberação de insulina. A dose insuficiente e a dose excessiva de insulina representam, na verdade, um único perigo, denominado de 'dose incorreta de insulina administrada', e pode ser elaborada uma única árvore de defeitos. Naturalmente, ao especificar como o software deve reagir aos perigos, você precisa distinguir entre a dose insuficiente e a excessiva de insulina.

A árvore de defeitos da Figura 9.3 está incompleta. Apenas os defeitos potenciais de software foram totalmente decompostos. Os defeitos de hardware, como baixa carga da bateria, que causam falha no sensor, não foram mostrados. Nesse nível, não é possível detalhar mais a análise. Contudo, à medida que um projeto e sua implementação são desenvolvidos, árvores de defeitos mais detalhadas podem ser desenvolvidas. Leveson e Harvey (Leveson e Harvey, 1983) e Leveson (Leveson, 1985) mostram como as árvores de defeitos podem ser desenvolvidas ao longo do projeto de software, até o nível de declarações individuais de linguagem de programação.

As árvores de defeitos são também usadas para identificar problemas potenciais de hardware. Uma árvore de defeitos pode fornecer percepções de requisitos de software para detectar e, talvez, corrigir esses problemas. Por exemplo, as doses de insulina não são administradas em uma freqüência muito alta, não mais que duas ou três vezes por hora, e às vezes menos

Figura 9.3
Árvore de defeitos do sistema de liberação de insulina.



freqüentemente do que isso. Portanto, a capacidade do processador fica disponível para executar programas de diagnóstico e autoverificação. Erros de hardware, como erros de sensor, bomba ou relógio, podem ser descobertos e comunicados antes que tragam conseqüências graves ao paciente.

9.1.4 Avaliação de redução de riscos

Após a identificação dos riscos potenciais e suas causas de origem, você precisa derivar os requisitos de confiabilidade do sistema, os quais gerenciam os riscos e asseguram que incidentes ou acidentes não ocorram. Existem três possíveis estratégias que você pode usar:

1. *Prevenção de riscos*. O sistema é projetado de modo que o risco ou o perigo não ocorra.
2. *Detecção e remoção de riscos*. O sistema é projetado de modo que os riscos sejam detectados e neutralizados antes que resultem em um acidente.
3. *Limitação de danos*. O sistema é projetado de modo que as conseqüências de um acidente sejam minimizadas.

Normalmente, os projetistas de sistemas críticos usam uma combinação dessas abordagens. Em um sistema crítico de segurança, os perigos intoleráveis podem ser tratados por meio da minimização de sua probabilidade e pelo acréscimo de um sistema de proteção que forneça um back-up de segurança. Por exemplo, em um sistema de controle de indústria química, o sistema tentará detectar e evitar o excesso de pressão no reator. Entretanto, deve haver também um sistema de proteção independente que monitore a pressão e abra uma válvula de alívio, caso seja detectada uma alta pressão.

No sistema de deliberação de insulina, um 'estado seguro' é o estado desligado, quando nenhuma insulina é injetada. Durante um curto período, isso não oferecerá uma ameaça à saúde do diabético. Se os problemas potenciais de software, identificados na Figura 9.3, forem considerados, as seguintes 'soluções' podem ser desenvolvidas:

Quadro 9.1 Exemplos de requisitos de segurança para uma bomba de insulina

RS1:	O sistema não deve fornecer uma dose única de insulina maior do que a dose máxima especificada para um usuário do sistema.
RS2:	O sistema não deve fornecer uma dose diária cumulativa de insulina maior do que a dose máxima especificada para um usuário do sistema.
RS3:	O sistema deve incluir um recurso de diagnóstico de hardware executado pelo menos quatro vezes por hora.
RS4:	O sistema incluirá um manipulador de exceções para todas as exceções identificadas na Tabela 3.
RS5:	Um alarme audível será emitido quando qualquer anomalia de hardware ou software for descoberta e uma mensagem de diagnóstico, conforme definida na Tabela 4, deverá ser exibida.
RS6:	No caso de acionamento do alarme, o fornecimento de insulina será suspenso até que o usuário reinicie o sistema e desative o alarme.

1. *Erro de aritmética*. Isso ocorre quando algum cálculo aritmético causa uma falha de representação. A especificação deve identificar todos os possíveis erros aritméticos que podem ocorrer. Esses erros dependem do algoritmo usado. A especificação pode estabelecer que um tratador de exceções deva ser incluído para cada erro aritmético identificado. A especificação deve estabelecer a ação a ser tomada em relação a cada um desses erros, caso ocorram. Uma ação segura é desligar o sistema de liberação e ativar um alarme de advertência.
2. *Erro de algoritmo*. Essa é uma situação mais difícil, pois nenhuma situação anômala definida pode ser detectada. Ela poderia ser detectada por meio da comparação do cálculo de uma dose necessária de insulina com a dose liberada anteriormente. Caso seja muito maior, isso pode significar que a quantidade foi calculada incorretamente. O sistema pode também manter o acompanhamento da seqüência de doses. Depois que um número de doses acima da média tiver sido liberado, um alarme poderá ser emitido e as dosagens seguintes poderão ser limitadas.

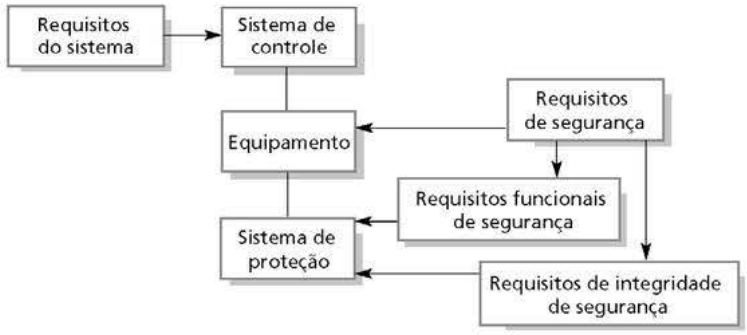
Alguns dos requisitos de segurança resultantes do sistema de bomba de insulina são mostrados no Quadro 9.1. Eles são requisitos de usuário e, naturalmente, seriam expressos com mais detalhes em uma especificação final do sistema. Nesses requisitos, as referências às tabelas 3 e 4 são relativas às tabelas que seriam incluídas no documento de requisitos.

9.2 Especificação de segurança

O processo de gerenciamento de riscos explicado até aqui evoluiu dos processos desenvolvidos para sistemas críticos de segurança. Até recentemente, sistemas críticos de segurança eram, em sua maioria, sistemas de controle, nos quais a falha de um equipamento que estivesse sendo controlado poderia causar danos. Nas décadas de 1980 e 1990, com o uso de controle por computador tornando-se amplo, a comunidade de engenharia de segurança desenvolveu padrões para especificação e desenvolvimento de sistemas críticos de segurança.

O processo de especificação e garantia de segurança é parte de um ciclo de vida geral de segurança definido em um padrão internacional para o gerenciamento de segurança, o IEC 61508 (IEC, 1998). Esse padrão foi desenvolvido especificamente para sistemas de proteção, como um sistema que pára um trem caso ele ultrapasse um sinal vermelho. Embora o padrão possa ser usado para sistemas críticos de segurança mais genéricos, como sistemas de controle, considero que sua separação da especificação de segurança baseada na especificação mais genérica do sistema não seja apropriada para sistemas críticos de informações. A Figura 9.4 ilustra o modelo de sistema que é suposto pelo padrão IEC 61508.

Figura 9.4
Requisitos de segurança de sistema de controle.



A Figura 9.5 é uma forma simplificada da representação de Redmill do ciclo de vida de segurança (Redmill, 1998). Como você pode verificar na Figura 9.5, o padrão aborda todos os aspectos do gerenciamento de segurança, desde a definição de escopo inicial, passando pelo planejamento e desenvolvimento de sistema, até sua desativação.

Nesse modelo, o sistema controla algum equipamento com requisitos de segurança de alto nível associados. Esses requisitos de alto nível geram dois tipos de requisitos de segurança mais detalhados que se aplicam ao sistema de proteção para o equipamento:

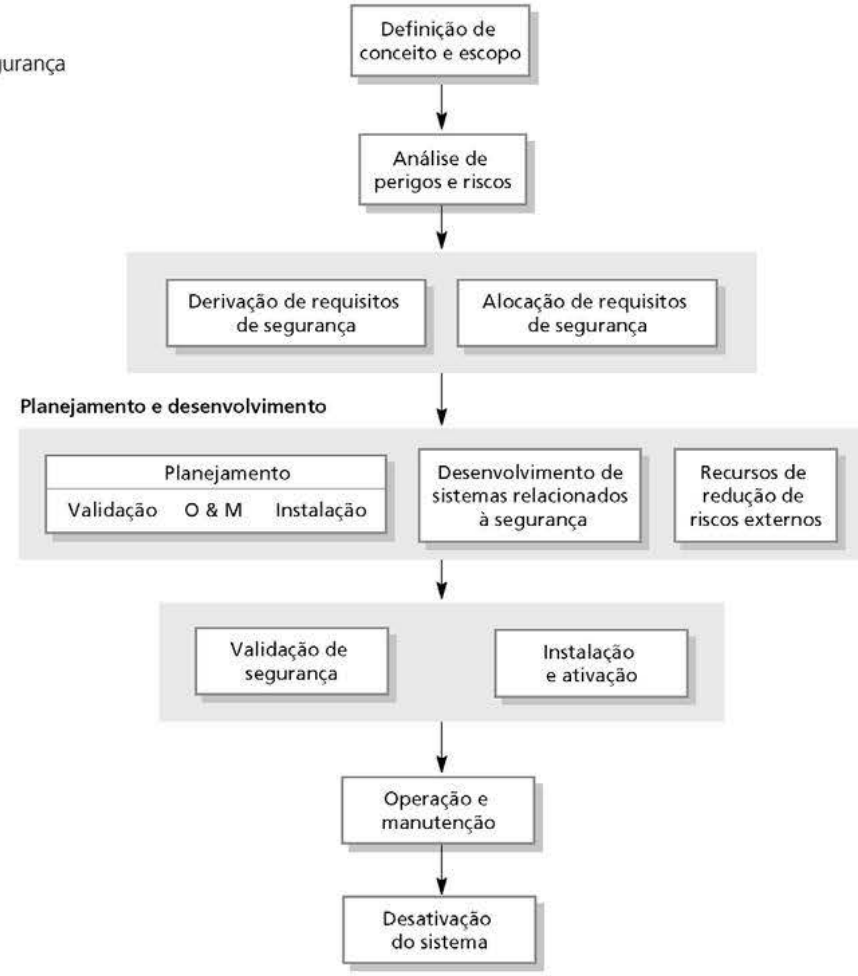
1. *Requisitos funcionais de segurança* que definem as funções de segurança do sistema.
2. *Requisitos de integridade de segurança* que definem a confiabilidade e a disponibilidade do sistema de proteção. Esses requisitos são baseados no uso esperado do sistema de proteção e destinam-se a assegurar que ele funcione quando necessário. Os sistemas são classificados por meio de um nível de integridade de segurança (SIL — Safety Integrity Level) de 1 a 4. Cada nível SIL representa um nível mais alto de confiabilidade; quanto mais crítico for o sistema, mais alto será o SIL exigido.

Os primeiros estágios do ciclo de vida de segurança do IEC 61508 definem o escopo do sistema, avaliam os perigos potenciais do sistema e estimam os riscos que eles causam. Isso é seguido pela especificação de requisitos de segurança e pela alocação desses requisitos aos diferentes subsistemas. A atividade de desenvolvimento envolve o planejamento e a implementação. O sistema crítico de segurança é projetado e implementado, assim como os sistemas externos que podem fornecer proteção adicional. Paralelamente, são planejadas a validação de segurança, a instalação e a operação e manutenção do sistema.

O gerenciamento de segurança não termina com a entrega do sistema. Depois da entrega, o sistema deve ser instalado conforme planejado, de modo que a análise de perigos permaneça válida. Portanto, a validação de segurança é realizada antes que o sistema seja colocado em operação. A segurança deve também ser gerenciada durante a operação e, particularmente, durante a manutenção do sistema. Muitos problemas relacionados à segurança surgem devido a um processo inadequado de

Figura 9.5

Ciclo de vida de segurança do IEC 61508.



manutenção e, por isso, é importante que o sistema seja projetado para facilitar a manutenção. Finalmente, as considerações de segurança que podem ser aplicadas durante a desativação (por exemplo, destruição de material perigoso em placas de circuito) devem ser levadas em consideração.

9.3 Especificação de proteção

A especificação de requisitos de proteção para sistemas tem algo em comum com os requisitos de segurança. Não é prático especificá-los quantitativamente, além de serem freqüentemente requisitos do tipo 'não deve' que definem um comportamento inaceitável do sistema em vez de uma funcionalidade necessária. No entanto, existem diferenças importantes entre esses tipos de requisitos:

1. A noção de um ciclo de vida de segurança que abranja todos os aspectos do gerenciamento de segurança está bem desenvolvida. Por outro lado, a área de especificação e gerenciamento de proteção é ainda imatura e não existe um ciclo de vida aceito.
2. Embora algumas ameaças à proteção sejam específicas de cada sistema, muitas são comuns em todos os tipos de sistema. Todos os sistemas devem se proteger contra intrusos, negação de serviços etc. Ao contrário, os perigos em sistemas críticos de segurança são específicos do domínio.
3. As técnicas e tecnologias de proteção, como dispositivos de criptografia e autenticação, estão bem maduras. Entretanto, o uso eficiente dessa tecnologia requer, freqüentemente, uma sofisticação técnica de alto nível. Pode haver dificuldade em instalar, configurar e atualizar. Conseqüentemente, os gerentes de sistema cometem erros, deixando vulnerabilidades no sistema.
4. A liderança de um fornecedor de software no mercado mundial significa que um grande número de sistemas podem ser afetados caso a proteção de seus programas apresente brechas. Não existe diversidade suficiente na infra-estrutura computacional e, conseqüentemente, ela é mais vulnerável a ameaças externas. Os sistemas críticos de segurança geralmente são especializados e feitos sob encomenda; sendo assim, essa situação não ocorre.

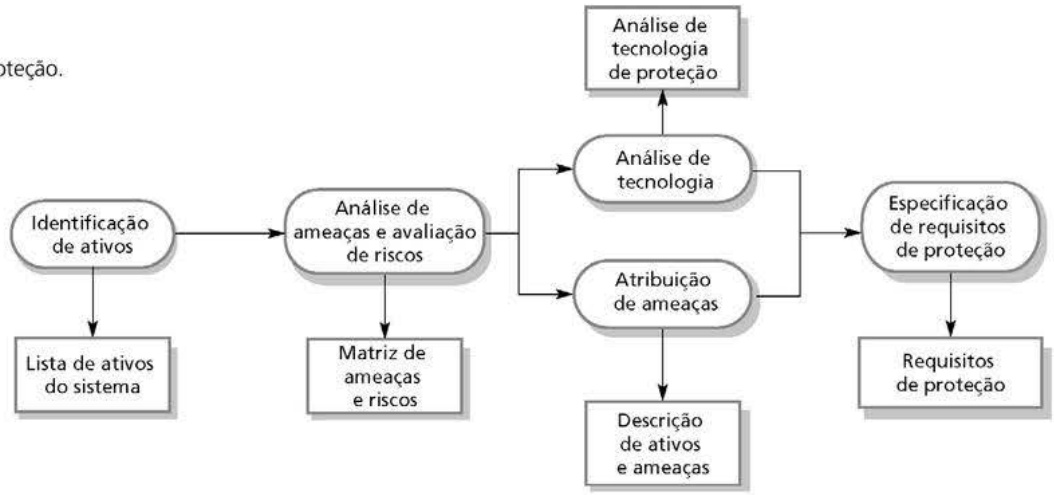
A abordagem convencional (não computadorizada) para a análise de proteção é baseada nos ativos a serem protegidos e em seu valor para uma organização. Portanto, um banco fornecerá alta proteção a uma área em que grandes quantidades de dinheiro são armazenadas, em comparação a outras áreas públicas, por exemplo, em que as perdas potenciais são limitadas. A mesma abordagem pode ser usada para a especificação de proteção para sistemas baseados em computadores. Um processo possível de especificação de proteção é mostrado na Figura 9.6.

Os estágios neste processo são:

1. *Identificação e avaliação de ativos*. Os ativos (dados e programas) e seus níveis necessários de proteção são identificados. A proteção necessária depende do valor do ativo e, assim, um arquivo de senhas, por exemplo, normalmente é mais valioso do que um conjunto de páginas Web públicas, uma vez que um ataque bem-sucedido ao arquivo de senhas traria conseqüências graves a todo o sistema.
2. *Análise de ameaças e avaliação de riscos*. Possíveis ameaças de proteção são identificadas e os riscos associados a cada uma dessas ameaças são estimados.

Figura 9.6

Especificação de proteção.



Quadro 9.2 Alguns requisitos de proteção do sistema LIBSYS

RP1:	Todos os usuários do sistema devem ser identificados por meio de seu número de cartão de biblioteca e senha pessoal.
RP2:	Os privilégios dos usuários devem ser atribuídos de acordo com sua classe (estudante, pessoal, pessoal de biblioteca).
RP3:	Antes da execução de qualquer comando, o LIBSYS deve verificar se o usuário tem privilégios suficientes para acessar e executar esse comando.
RP4:	Quando um usuário pedir um documento, a solicitação do pedido deverá ser registrada. Os dados de registro mantidos incluirão a hora do pedido, a identificação do usuário e os artigos pedidos.
RP5:	Todos os dados do sistema devem ser gravados uma vez por dia, e os back-ups, armazenados em uma área de repositório protegida, fora do local do sistema.
RP6:	Não deve ser permitido que os usuários efetuem mais de um login simultâneo no LIBSYS.

3. *Atribuição de ameaças*. As ameaças identificadas são relacionadas aos ativos, de modo que, para cada ativo identificado, exista uma lista de ameaças associadas.
4. *Análise de tecnologia*. As tecnologias disponíveis de proteção e a sua aplicabilidade são avaliadas em relação às ameaças identificadas.
5. *Especificação de requisitos de proteção*. Os requisitos de proteção são especificados. Quando apropriados, eles identificam explicitamente as tecnologias que podem ser usadas para proteção contra as ameaças ao sistema.

A especificação e o gerenciamento de proteção são essenciais a todos os sistemas críticos. Se um sistema não for seguro, então ele estará sujeito à infecção por vírus e worms, corrompimento e modificação não autorizada de dados, e aos ataques de negação de serviços. Isso significa que não podemos estar confiantes de que os esforços para assegurar a segurança e a confiabilidade serão eficientes.

Diferentes tipos de requisitos de proteção indicam as diferentes ameaças enfrentadas por um sistema. Firesmith (Firesmith, 2003) identifica dez tipos de requisitos de proteção que podem ser incluídos em um sistema:

1. *Requisitos de identificação* especificam se um sistema deve identificar seus usuários antes de interagir com eles.
2. *Requisitos de autenticação* especificam como os usuários são identificados.
3. *Requisitos de autorização* especificam os privilégios e as permissões de acesso de usuários identificados.
4. *Requisitos de imunidade* especificam como um sistema deve se proteger contra vírus, worms e ameaças similares.
5. *Requisitos de integridade* especificam como o corrompimento de dados pode ser evitado.
6. *Requisitos de detecção de intrusão* especificam quais mecanismos devem ser usados para detectar ataques ao sistema.
7. *Requisitos de não-rejeição* especificam que uma parte de uma transação não pode negar seu envolvimento nessa transação.
8. *Requisitos de privacidade* especificam como a privacidade dos dados será mantida.
9. *Requisitos de auditoria de proteção* especificam como o uso de sistema pode ser auditado e verificado.
10. *Requisitos de proteção de manutenção do sistema* especificam como uma aplicação pode impedir mudanças autorizadas devido à falha acidental em seus mecanismos de proteção.

Naturalmente, nem todos os sistemas necessitam de todos esses requisitos de proteção. Requisitos específicos dependem do tipo de sistema, da situação de uso e dos usuários esperados. Como exemplo, o Quadro 9.2 mostra os requisitos de proteção que devem ser incluídos no sistema LIBSYS.

9.4 Especificação de confiabilidade de software

Confiabilidade é um conceito complexo que deve sempre ser considerado no nível de sistema e não no nível de componentes individuais. Como os componentes em um sistema são interdependentes, uma falha em um componente pode se propagar pelo sistema e afetar a operação de outros componentes. Em um sistema baseado em computadores, você precisa considerar três dimensões ao especificar a confiabilidade global do sistema:

1. *Confiabilidade do hardware*. Qual é a probabilidade de um componente de hardware falhar e quanto tempo levaria para reparar esse componente?

2. *Confiabilidade do software*. Qual é a probabilidade de um componente de software produzir uma saída incorreta? As falhas de software são diferentes das falhas de hardware, pois o software não se desgasta. Ele pode continuar operando corretamente depois de produzir um resultado incorreto.
3. *Confiabilidade do operador*. Qual é a probabilidade de o operador de um sistema cometer um erro?

Todas essas dimensões estão fortemente ligadas. A falha de hardware pode causar sinais espúrios fora da faixa de entradas esperadas pelo software. O software pode, então, se comportar de maneira imprevisível. O comportamento inesperado do sistema pode confundir e causar estresse no operador. O operador pode, por isso, agir incorretamente e escolher entradas não apropriadas para a situação atual de falha. Essas entradas podem confundir mais o sistema, e mais erros serão gerados. Uma única falha de subsistema recuperável pode, dessa maneira, desenvolver rapidamente um problema grave que necessite de um desligamento completo do sistema.

A confiabilidade de sistemas deve ser especificada como um requisito não funcional que, de modo ideal, é expresso quantitativamente, usando uma das métricas explicadas na próxima seção. Para atender aos requisitos não funcionais de confiabilidade, pode ser necessário especificar mais requisitos funcionais e de projeto, que especificam como as falhas podem ser evitadas ou toleradas. Exemplos desses requisitos de confiabilidade são:

1. Deve haver uma faixa predefinida de todos os valores inseridos pelo operador e o sistema deve verificar se todas as entradas do operador estão dentro dessa faixa predefinida.
2. Como parte do processo de inicialização, o sistema deve verificar todos os discos procurando por blocos defeituosos.
3. A programação em N-versões deve ser usada para implementar o sistema de controle de freios.
4. O sistema deve ser implementado em um subconjunto seguro da linguagem Ada verificado por meio de análise estática.

Não existem regras simples para derivar os requisitos funcionais de confiabilidade. Em organizações que desenvolvem sistemas críticos, existe geralmente conhecimento organizacional sobre possíveis requisitos de confiabilidade e como eles afetam a confiabilidade real de um sistema. Essas organizações podem se especializar em tipos específicos de sistemas, como sistemas de controle ferroviário, de modo que os requisitos de confiabilidade, uma vez derivados, são reusados por vários sistemas. Quanto mais alto for o nível de integridade de segurança (explicado anteriormente) necessário em sistemas críticos de segurança, mais rigorosos serão, provavelmente, os requisitos de confiabilidade.

9.4.1 Métricas de confiabilidade

As métricas de confiabilidade foram inicialmente criadas para componentes de hardware. Uma falha em um componente de hardware é inevitável devido aos fatores físicos como desgaste mecânico e aquecimento elétrico. Os componentes têm tempo de vida limitado e isso reflete-se na métrica de confiabilidade de hardware mais amplamente usada, que é o tempo médio para falha (MTTF — Mean Time To Failure). O MTTF é o tempo médio no qual se espera que o componente esteja operacional. A falha do componente de hardware geralmente é permanente e, assim, o tempo médio para reparo (MTTR — Mean Time To Repair), que reflete o tempo necessário para reparar ou substituir o componente, é também significativo.

Entretanto, essas métricas de hardware não são diretamente aplicáveis à especificação de confiabilidade de software, pois as falhas de componentes de software são, muitas vezes, transitórias e não permanentes. Elas aparecem somente em algumas entradas. Se os dados não estiverem danificados, em geral o sistema poderá continuar em operação após a ocorrência de uma falha.

As métricas usadas para a especificação de confiabilidade e disponibilidade de software são mostradas na Tabela 9.2. A escolha de qual métrica deve ser usada depende do tipo de sistema ao qual se aplica e dos requisitos do domínio da aplicação. Alguns exemplos de tipos de sistema nos quais essas diferentes métricas podem ser usadas são:

1. *Probabilidade de falha sob demanda (POFOD)*. Essa métrica é mais apropriada para sistemas em que os serviços são solicitados em intervalos de tempo não previstos ou em intervalos relativamente longos e nos quais existem conseqüências graves, caso o serviço não seja prestado. Ela poderia ser usada para especificar sistemas de proteção, como a confiabilidade de um sistema de alívio de pressão em uma indústria química ou um sistema de desligamento de emergência em uma usina nuclear.
2. *Taxa de ocorrência de falhas (ROCOF)*. Essa métrica deve ser usada quando forem feitas solicitações regulares de serviços do sistema e quando for importante que esses serviços sejam prestados corretamente. Ela poderia ser usada na especificação de um sistema de caixa eletrônico bancário, que processa as transações de clientes, ou no sistema de reservas de um hotel.
3. *Tempo médio para falha (MTTF)*. Essa métrica deve ser usada em sistemas nos quais existem transações longas, ou seja, nos quais as pessoas usam o sistema por um longo tempo. O MTTF deve ser mais longo do que o tempo

Tabela 9.2 Métricas de confiabilidade

Métrica	Explicação
POFOD Probabilidade de falha sob demanda	A probabilidade de que o sistema falhará quando for feita uma solicitação de serviço. Uma POFOD de 0,001 significa que uma entre mil solicitações de serviços pode resultar em falha.
ROCOF Taxa de ocorrência de falhas	A freqüência com que um comportamento inesperado pode ocorrer. Uma ROCOF de 2/100 significa que é provável que ocorram duas falhas em cada 100 unidades operacionais de tempo. Essa métrica é, às vezes, chamada de intensidade de falhas.
MTTF Tempo médio para falhar	O tempo médio entre as falhas observadas no sistema. Um MTTF de 500 significa que se pode esperar uma falha a cada 500 unidades de tempo.
AVAIL Disponibilidade	A probabilidade de que o sistema esteja disponível para uso em um dado tempo. Uma disponibilidade de 0,998 significa que o sistema provavelmente estará disponível em 998 de cada 1.000 unidades de tempo.

médio de cada transação. Exemplos de sistemas em que essa métrica pode ser usada são sistemas de processamento de texto e sistemas CAD.

4. *Disponibilidade (AVAIL).* Essa métrica deve ser usada para sistemas 'non-stop', nos quais os usuários esperam que o sistema preste um serviço contínuo. Exemplos desses sistemas são os sistemas de comutação telefônica e sistemas de sinalização ferroviária.

Existem três tipos de medições que podem ser realizadas quando é avaliada a confiabilidade de um sistema:

1. O número de falhas dado um número de solicitações de serviços de sistema. É usada para medir a POFOD.
2. O tempo (ou número de transações) entre as falhas do sistema. É usada para medir a ROCOF e o MTTF.
3. O tempo decorrido para reparar ou para reiniciar, após a ocorrência de uma falha de sistema. Dado que o sistema deve estar continuamente disponível, é usado para medir a AVAIL.

As unidades de tempo que podem ser usadas nessas métricas são tempo de calendário, tempo de processador ou alguma unidade discreta, como o número de transações. Em sistemas que passam grande parte do tempo aguardando para responder a uma solicitação de serviço, como sistemas de comutação telefônica, a unidade de tempo que deve ser usada é o tempo de processador. Se você usar o tempo de calendário, será incluído o tempo em que o sistema estava ocioso.

O tempo de calendário é uma unidade de tempo apropriada para sistemas que estejam em operação contínua. Por exemplo, os sistemas de monitoração, como sistemas de alarme e outros tipos de sistemas de controle de processo, estão nessa categoria. Os sistemas que processam transações, como caixas eletrônicos bancários e sistema de reserva de passagens aéreas, têm cargas variáveis dependendo da hora do dia. Nesses casos, a unidade de 'tempo' usada deve ser o número de transações, isto é, a ROCOF seria o número de transações que falharam por N milhares de transações.

9.4.2 Requisitos não funcionais de confiabilidade

Em vários documentos de requisitos de sistema, os requisitos de confiabilidade não são especificados cuidadosamente. As especificações de confiabilidade são subjetivas e não mensuráveis. Por exemplo, as declarações do tipo 'O software deve ser confiável sob condições normais de uso' não têm significado. As declarações quase quantitativas do tipo 'O software deve exibir não mais que N defeitos/1.000 linhas' também são igualmente inúteis. É impossível medir o número de defeitos/1.000 linhas de código, assim não é possível dizer quando todos os defeitos foram descobertos. Além disso, a declaração não significa nada em termos do comportamento dinâmico do sistema. São as falhas de software, não seus defeitos, que afetam a confiabilidade do sistema.

Os tipos de falhas que podem ocorrer são específicas de sistema, e as conseqüências de uma falha de sistema dependem de sua natureza. Ao escrever uma especificação de confiabilidade, você deve identificar diferentes tipos de falhas e pensar se eles devem ser tratados de maneira diferenciada na especificação. Exemplos de tipos diferentes de falhas são mostrados na Tabela 9.3. Obviamente, podem ocorrer combinações dessas falhas, como uma que seja transitória, recuperável e de corrompimento.

A maioria dos sistemas de grande porte é composta de diversos subsistemas com diferentes requisitos de confiabilidade. Como um software com nível muito alto de confiabilidade é muito oneroso, você deve avaliar os requisitos de confiabilidade de cada subsistema separadamente, em vez de impor o mesmo requisito de confiabilidade a todos os subsistemas. Isso evita impor altas exigências de confiabilidade sobre esses subsistemas sem necessidade.

Tabela 9.3 Classificação de falhas

Tipo de falha	Descrição
Transitória	Ocorre apenas com determinadas entradas.
Permanente	Ocorre com todas as entradas.
Recuperável	O sistema pode se recuperar sem intervenção do operador.
Irrecuperável	A intervenção do operador é necessária para a recuperação da falha.
Não de corrompimento	A falha não corrompe o estado ou os dados do sistema.
De corrompimento	A falha corrompe o estado ou os dados do sistema.

As etapas envolvidas no estabelecimento de uma especificação de confiabilidade são:

1. Para cada subsistema, identificar os tipos de falhas de sistema que podem ocorrer e analisar as conseqüências dessas falhas.
2. Com base na análise de falhas no sistema, divida as falhas em classes apropriadas. Um ponto de partida razoável é usar os tipos de falhas apresentados na Tabela 9.3.
3. Para cada tipo de falha identificada, definir o requisito de confiabilidade usando uma métrica de confiabilidade adequada. Não é necessário usar a mesma métrica para diferentes tipos de falhas. Se uma falha exigir alguma intervenção para que o sistema se recupere, a probabilidade de ocorrência dessa falha sob demanda poderia ser a métrica mais apropriada. Quando a recuperação automática for possível e o efeito da falha não for conveniente para o usuário, a ROCOF pode ser mais apropriada.
4. Quando apropriado, identificar os requisitos funcionais de confiabilidade que definem a funcionalidade de sistema para reduzir a probabilidade de falhas críticas.

Como exemplo, considere os requisitos de confiabilidade de um caixa eletrônico bancário. Suponha que cada máquina na rede seja usada aproximadamente 300 vezes por dia. O tempo de vida do hardware do sistema é de cinco anos e o software é atualizado, normalmente, a cada ano. Portanto, durante o tempo de vida de uma release do software, cada máquina cuidou de cerca de 100.000 transações. Um banco tem 1.000 equipamentos em sua rede. Isso significa que existem cerca de 300.000 transações por dia no banco de dados central (digamos 100 milhões por ano).

A Tabela 9.4 mostra os possíveis tipos de falhas e as possíveis especificações de confiabilidade para diferentes tipos de falhas de sistema. Os requisitos de confiabilidade estabelecem como aceitável a ocorrência de falha permanente na máquina uma vez a cada três anos aproximadamente. Isso significa que, em média, uma máquina na rede bancária poderia ser afetada a cada dia. Ao contrário, os defeitos que significam que uma transação precisa ser cancelada podem ocorrer com relativa freqüência. Sua única conseqüência é causar pequena inconveniência ao usuário.

De modo ideal, os defeitos que corrompem o banco de dados nunca devem ocorrer no tempo de vida do software. Portanto, o requisito de confiabilidade que poderia ser definido é que a probabilidade de uma falha de corrompimento ocorrer, quando uma demanda é realizada, é menor do que 1 em 200 milhões de transações. Ou seja, durante o tempo de vida de uma release do software do caixa eletrônico, nunca deve ocorrer um erro que cause corrompimento do banco de dados.

No entanto, um requisito de confiabilidade como esse não pode realmente ser testado. Digamos que cada transação leve um segundo do tempo de máquina e um simulador possa ser construído para a rede de caixas eletrônicos. A simulação das

Tabela 9.4 Especificação de confiabilidade de um caixa eletrônico

Tipo de falha	Exemplo	Métrica de confiabilidade
Permanente, não de corrompimento	O sistema falha ao operar com qualquer cartão inserido. O software deve ser reiniciado para corrigir a falha.	ROCOF 1 ocorrência em 1.000 dias.
Transitória, não de corrompimento	Os dados da faixa magnética não podem ser lidos em um cartão sem danos inserido.	ROCOF 1 ocorrência em 1.000 transações.
Transitória, de corrompimento	Um padrão de transações na rede causa o corrompimento do banco de dados.	Não quantificável! Nunca deve ocorrer no tempo de vida do sistema.

transações que ocorrem em um único dia na rede levará 300.000 segundos. Isso corresponde a aproximadamente 3 dias e meio. Claramente, esse período poderia ser reduzido pela redução do tempo de transação e uso de vários simuladores. Contudo, é ainda muito difícil testar o sistema para validar a especificação de confiabilidade.

É impossível validar requisitos qualitativos que demandam um nível muito alto de confiabilidade. Por exemplo, digamos que um sistema destina-se ao uso em uma aplicação crítica de segurança e, portanto, nunca deve falhar ao longo de todo o tempo de vida do sistema. Suponha que 1.000 cópias do sistema estejam instaladas e o sistema seja 'executado' 1.000 vezes por segundo. O tempo de vida projetado do sistema é de 10 anos. O número total estimado de execuções de sistema será, portanto, aproximadamente, $3 \cdot 10^{14}$. Não existe um ponto na especificação de que a taxa de ocorrência de falha deva ser de $1/10^{15}$ execuções (isso permite algum fator de segurança) quando você não pode testar o sistema por tempo suficiente para validar esse nível de confiabilidade.

Como outro exemplo, considere os requisitos de confiabilidade para o sistema de bomba de insulina. Esse sistema libera insulina um número de vezes por dia e monitora a glicose do sangue do usuário várias vezes por hora. Como o uso do sistema é intermitente e as conseqüências de falhas são graves, a métrica de confiabilidade mais apropriada seria a POFOD (Probability Of Failure On Demand).

A falha em liberar insulina não tem implicações imediatas de segurança e, portanto, os fatores comerciais, e não os fatores de segurança, orientam o nível de confiabilidade necessário. Os custos de serviço são altos porque os usuários precisam de reparos e substituições rápidos. É de interesse do fabricante limitar o número de falhas permanentes que necessitem de reparo.

Novamente, dois tipos de falhas podem ser identificados:

1. *Falhas transitórias* que podem ser corrigidas pelas ações do usuário, como reiniciação ou calibração do equipamento. Para esses tipos de falhas, um valor relativamente baixo de POFOD (digamos 0,002) pode ser aceitável. Isso significa que pode ocorrer uma falha a cada 500 demandas feitas ao equipamento. Isso representa aproximadamente uma vez a cada 3 dias e meio.
2. *Falhas permanentes* que exigem que o equipamento seja consertado pelo fabricante. A probabilidade desse tipo de falha deve ser muito baixa. Aproximadamente uma vez por ano é o número mínimo e, então, a POFOD não deve ser maior do que 0,00002.

O custo de desenvolvimento e validação de uma especificação de confiabilidade do sistema pode ser muito alto. As organizações devem ser realistas sobre se esses custos valem a pena. Os custos são claramente justificados em sistemas nos quais a operação confiável seja crítica, como sistemas de comutação telefônica, ou nos quais a falha do sistema possa resultar em grandes prejuízos financeiros. Eles não são, provavelmente, justificáveis para muitos tipos de sistemas de negócios ou científicos. Esses sistemas têm requisitos modestos de confiabilidade, sendo que os custos de falhas serão simplesmente atrasos no processamento, e é relativamente simples e barato se recuperar dessas falhas.

PONTOS-CHAVE

- A análise de riscos é uma atividade fundamental no processo de especificação de sistemas críticos. Ela envolve a identificação de riscos que podem resultar em acidentes ou incidentes. Os requisitos de sistema são, então, gerados para assegurar que os riscos não ocorram ou, se ocorrerem, não resultem em um incidente.
- A análise de riscos é o processo de avaliação de probabilidade de que um risco resultará em um acidente. A análise de riscos identifica os riscos críticos que devem ser evitados no sistema e classifica os riscos de acordo com sua gravidade.
- Para especificar os requisitos de proteção, você deve identificar os ativos protegidos e definir como as técnicas e a tecnologia de proteção devem ser usadas para proteger esses ativos.
- Requisitos de confiabilidade devem ser definidos quantitativamente na especificação de requisitos de sistema.
- Existem várias métricas de confiabilidade, como probabilidade de falha sob demanda (POFOD — Probability Of Failure On Demand), taxa de ocorrência de falhas, tempo médio para falha (MTTF — Mean Time To Failure) e disponibilidade. A métrica mais apropriada para um sistema específico depende do tipo de sistema e do domínio da aplicação. Métricas diferentes podem ser usadas por subsistemas diferentes.
- As especificações não funcionais de confiabilidade podem levar a requisitos funcionais de sistema que definam as características do sistema, cuja função é reduzir o número de falhas de sistema e, portanto, aumentar a confiabilidade.

LEITURAS SUGERIDAS

"Security use cases." Um bom artigo, disponível na Web, que enfoca como os casos de uso podem ser usados na especificação de proteção. O autor também tem uma série de bons artigos sobre especificação de proteção citados neste artigo. (D. G. Firesmith, *Journal of object technology,* 2 (3), maio/jun. de 2003.)

"Requirements definition for survivable network systems." Explica os problemas de definição de requisitos para sistemas de sobrevivência, nos quais a capacidade de sobrevivência esteja relacionada tanto com a disponibilidade quanto com a proteção (R. C. Linger, et al., *Proc. ICRE'98*, IEEE Press, 1998.)

Requirements engineering: a good practice guide. Este livro inclui uma seção sobre a especificação de sistemas críticos e uma explicação do uso de métodos formais na especificação de sistemas críticos. (I. Sommerville e P. Sawyer, 1997, John Wiley & Sons.)

Safeware: system safety and computers. Esta é uma explicação minuciosa de todos os aspectos de sistemas críticos de segurança. É particularmente rígido em sua descrição de análise de perigos e na derivação de requisitos. (N. Leveson, 1995, Addison-Wesley.)

EXERCÍCIOS

9.1 Explique por que os limites no triângulo de risco, mostrados na Figura 9.2, estão sujeitos a mudar com o tempo e com a mudança de atitudes sociais.

9.2 No sistema de bomba de insulina, o usuário precisa trocar a agulha e o fornecimento de insulina em intervalos regulares e pode também mudar a dose única máxima e a dose máxima diária administradas. Sugira três erros do usuário que podem ocorrer e proponha requisitos de segurança que evitem que esses erros resultem em acidente.

9.3 Um sistema de software crítico de segurança para tratamento de pacientes com câncer possui dois principais componentes:

- Um equipamento de terapia por radiação que libera doses controladas de radiação nos pontos onde existe tumor. Esse equipamento é controlado por um sistema de software embutido.
- Um banco de dados de tratamento que inclui detalhes do tratamento dado a cada paciente. Os requisitos de tratamento são inseridos nesse banco de dados e são automaticamente baixados para o equipamento de terapia por radiação.

Identifique três perigos que podem ocorrer nesse sistema. Para cada perigo, sugira um requisito defensivo que reduzirá a probabilidade de que esses perigos resultem em um acidente. Explique por que a defesa sugerida pode reduzir o risco associado ao perigo.

9.4 Descreva três diferenças importantes entre os processos de especificação de segurança e de especificação de proteção.

9.5 Sugira como a análise de árvores de defeitos poderia ser modificada para uso na especificação de proteção. Ameaças em um sistema crítico de proteção são análogos aos perigos em um sistema crítico de segurança.

9.6 Qual é a principal diferença entre falhas de hardware e de software? Dada essa diferença, explique por que as métricas de confiabilidade de hardware não são, geralmente, apropriadas para medir a confiabilidade de software.

9.7 Explique por que é praticamente impossível validar as especificações de confiabilidade, quando elas são expressas em termos de um número muito pequeno de falhas ao longo do tempo de vida total de um sistema.

9.8 Sugira métricas de confiabilidade apropriadas para as seguintes classes de sistemas de software. Justifique sua escolha de métricas. Preveja o uso desses sistemas e sugira valores apropriados para as métricas de confiabilidade.

- Um sistema que monitora pacientes em uma UTI hospitalar
- Um processador de texto
- Um sistema de controle de máquina de venda automática
- Um sistema para controlar os freios de um automóvel
- Um sistema para controlar um refrigerador
- Um gerador de relatórios gerenciais.

9.9 Você é responsável por escrever a especificação de um sistema de software que controla uma rede de terminais de pontos-de-venda eletrônicos em uma loja. O sistema aceita informações de código de barras de um terminal, consultas a um banco de dados de produtos, e retorna o nome do item e seu preço para serem exibidos no terminal. O sistema deve estar continuamente disponível durante o horário de funcionamento da loja.

Justificando sua escolha, selecione métricas apropriadas para especificar a confiabilidade de tal sistema e escreva uma especificação de confiabilidade convincente que leve em consideração o fato de que alguns defeitos são mais graves do que outros.

Sugira quatro requisitos funcionais que podem ser gerados para esse sistema de loja para aumentar a confiabilidade do sistema.

9.10 Um sistema de proteção de trem aciona automaticamente os freios de um trem caso o limite de velocidade em um trecho da via seja excedido ou se o trem entrar em um trecho exibindo um sinal vermelho (ou seja, o trecho não deveria estar ocupado). Justificando sua resposta, selecione uma métrica de confiabilidade que possa ser usada para especificar a confiabilidade necessária para esse sistema.

Existem dois requisitos essenciais de segurança para esse sistema:

- O trem não deve entrar no trecho da via indicado com sinal vermelho.
- O trem não deve exceder o limite de velocidade especificado para um trecho da via.

Supondo que o status do sinal e o limite de velocidade para o trecho da via são transmitidos para o computador de bordo do trem antes da sua entrada no trecho da via, proponha cinco requisitos funcionais de sistema possíveis para o software de bordo que possam ser gerados baseados nos requisitos de segurança de sistema.

9.11 Os engenheiros de software que trabalham com a especificação e o desenvolvimento de sistemas relacionados à segurança devem ter algum tipo de certificação profissional? Justifique sua resposta.

9.12 Como especialista em proteção de computador, você foi procurado por uma organização que faz campanha pelos direitos das vítimas de tortura para ajudá-los a obter acesso não autorizado aos sistemas de computadores de uma empresa britânica. Isso irá ajudá-los a confirmar ou não que essa empresa está vendendo equipamentos usados diretamente na tortura de presos políticos. Explique os dilemas éticos levantados por essa solicitação e como você reagiria nesse caso.

10

Especificação formal

Objetivos

O objetivo deste capítulo é apresentar técnicas de especificação formal que podem ser usadas para acrescentar detalhes a uma especificação de requisitos de sistema. Após ler este capítulo, você compreenderá:

- por que as técnicas de especificação formal auxiliam na descoberta de problemas em requisitos de sistema;
- o uso de técnicas algébricas da especificação formal para definir especificações de interface;
- como as técnicas formais e baseadas em modelos são usadas para especificação de comportamento.

Conteúdo

10.1 Especificação formal no processo de software
10.2 Especificação de interface de subsistema
10.3 Especificação de comportamento

Nas disciplinas de engenharia 'tradicionais', como a engenharia elétrica e a civil, o progresso geralmente envolve o desenvolvimento de melhores técnicas matemáticas. O setor de engenharia não tem tido dificuldade em aceitar a necessidade da análise matemática e em incorporá-la em seus processos. A análise matemática é uma parte rotineira do processo de desenvolvimento e validação de um projeto de produto.

No entanto, a engenharia de software não seguiu os mesmos passos. Embora haja mais de 30 anos de pesquisas de uso de técnicas matemáticas no processo de software, essas técnicas têm tido um impacto limitado. Os assim chamados métodos formais de desenvolvimento de software não são amplamente usados no desenvolvimento de software industrial. A maioria das empresas de desenvolvimento de software não os considera adequados quanto ao custo para aplicá-los em seus processos de desenvolvimento de software.

O termo *métodos formais* é usado para indicar quaisquer atividades que contam com representações matemáticas de software, entre elas a especificação formal de sistema, análise e prova de especificação, desenvolvimento transformacional e verificação de programa. Todas essas atividades dependem da especificação formal de software. Uma especificação formal de software é uma especificação expressa em uma linguagem cujos vocabulário, sintaxe e semântica são formalmente definidos. Essa necessidade de uma definição formal significa que as linguagens de especificação devem ser baseadas em conceitos matemáticos, cujas propriedades estejam bem compreendidas. O ramo da matemática usado é a matemática discreta e os conceitos matemáticos provêm da teoria dos conjuntos, lógica e álgebra.

Na década de 1980, muitos pesquisadores de engenharia de software apresentam a hipótese de que o uso de métodos formais de desenvolvimento era o melhor modo de aprimorar a qualidade do software. Eles argumentavam que o rigor e a análise detalhada, que constituem a principal parte dos métodos formais, levariam a programas com menos erros e que fossem mais adequados às necessidades dos usuários. Eles previram que, no século XXI, grande parte do software seria desenvolvida com o uso de métodos formais.

Essa previsão evidentemente não se concretizou. Existem quatro razões principais para isso:

1. *Engenharia de software bem-sucedida.* O uso de métodos de engenharia de software como métodos estruturados, gerenciamento de configuração e ocultação de informações nos processos de projeto e desenvolvimento de software resultaram no aprimoramento da qualidade do software. As pessoas que sugeriram que a única maneira de aprimorar a qualidade do software era o uso de métodos formais estavam claramente enganadas.
2. *Mudança de mercado.* Na década de 1980, a qualidade de software era vista como o problema principal da engenharia de software. Entretanto, desde então, o ponto crítico de várias categorias de desenvolvimento de software não tem sido a qualidade, mas o tempo de entrega do produto. O software deve ser desenvolvido rapidamente, e os clientes, às vezes, aceitam o software com alguns defeitos se a entrega rápida puder ser realizada. As técnicas para desenvolvimento rápido de software não funcionam eficazmente com especificações formais. Naturalmente, a qualidade ainda é um fator importante, mas deve ser obtida no contexto de entrega rápida.
3. *Escopo limitado dos métodos formais.* Os métodos formais não são muito adequados para especificar as interfaces e interação com o usuário. O componente de interface com o usuário tem se tornado, cada vez mais, a parte mais significativa da maioria dos sistemas e, portanto, os métodos formais só podem ser usados de fato no desenvolvimento de outras partes do sistema.
4. *Escalabilidade limitada dos métodos formais.* Os métodos formais ainda não são tão escaláveis. Os projetos bem-sucedidos que usaram essas técnicas foram, em sua maior parte, relativos a sistemas críticos de kernel de porte relativamente pequeno. À medida que aumenta o tamanho do sistema, o tempo e o esforço necessários para desenvolver uma especificação formal crescem desproporcionalmente.

Esses fatores indicam que a maioria das empresas de desenvolvimento de software não tem desejado arriscar-se no uso dos métodos formais em seu processo de desenvolvimento. Contudo, a especificação formal é uma excelente maneira para descobrir erros de especificação e apresentar a especificação do sistema de modo não ambíguo. As organizações que têm feito investimentos em métodos formais têm constatado menos erros no software entregue, sem aumento nos custos de desenvolvimento. Parece que os métodos formais podem ser adequados em custo caso seu uso seja limitado a partes do núcleo do sistema e caso as empresas desejem fazer alto investimento inicial nessa tecnologia.

O uso dos métodos formais está aumentando na área de desenvolvimento de sistemas críticos, na qual as propriedades emergentes de sistema, tais como segurança, confiabilidade e proteção, são muito importantes. O alto custo de falha nesses sistemas significa que as empresas desejam aceitar os altos custos iniciais dos métodos formais para assegurar que o seu software seja tão confiável quanto possível. Conforme discuto no Capítulo 24, os sistemas críticos têm custos muito altos de validação, e os custos da falha de sistema são altos e estão aumentando. Os métodos formais podem reduzir os custos.

Os sistemas críticos nos quais os métodos formais foram aplicados com sucesso incluem um sistema de controle de informações de tráfego aéreo (Hall, 1996), sistemas de sinalização de ferrovia (Dehbonei e Mejia, 1995), sistemas aeroespaciais (Easterbrook, et al., 1998) e sistemas de controle médico (Jacky, et al., 1997; Jacky, 1995). Eles também foram usados para especificação de ferramentas de software (Neil, et al., 1998), para especificação do sistema CICS da IBM (Wordsworth, 1991) e para o núcleo de um sistema de tempo real (Spivey, 1990). O método Cleanroom de desenvolvimento de software (Prowell, et al., 1999) baseia-se formalmente em argumentos de que o código está em conformidade com sua especificação. Como é também possível pensar na proteção de um sistema por meio do desenvolvimento de uma especificação formal, é possível que sistemas de proteção sejam uma área importante para o uso de métodos formais (Hall e Chapman, 2002).

10.1 Especificação formal no processo de software

O desenvolvimento de sistemas críticos geralmente envolve um processo de software baseado em planos que, por sua vez, baseia-se no modelo de desenvolvimento em cascata, explicado no Capítulo 4. Tanto os requisitos quanto o projeto de sistema são analisados e verificados cuidadosamente antes do início da implementação. Se uma especificação formal do software for desenvolvida, ela virá depois que os requisitos do sistema forem especificados, porém antes do projeto detalhado do sistema. Existe um loop estreito de realimentação entre a especificação detalhada de requisitos e a especificação formal. Conforme explicarei mais adiante, um dos principais benefícios da especificação formal é a capacidade de descobrir problemas e ambigüidades nos requisitos do sistema.

O envolvimento do cliente diminui e o envolvimento do fornecedor aumenta à medida que mais detalhes são acrescentados à especificação do sistema. Nos estágios iniciais do processo, a especificação deve ser 'orientada ao cliente'. Você deve escrever a especificação de modo que o cliente possa entender e deve ser formulado o mínimo possível de hipóteses sobre o projeto de software. Contudo, o estágio final do processo, que é a construção de uma especificação completa, consistente e precisa, destina-se principalmente ao fornecedor de software. Esse documento especifica os detalhes da implementação do sistema. Você pode usar uma linguagem formal nesse estágio para evitar ambigüidade na especificação de software.

A Figura 10.1 mostra os estágios da especificação de software e sua interface com o processo de projeto. Os estágios de especificação mostrados na Figura 10.1 não são independentes nem necessariamente desenvolvidos na seqüência apresentada. A Figura 10.2 mostra as atividades de especificação e projeto que podem ser realizadas em fluxos paralelos. Existe um relacionamento de duas vias entre cada estágio do processo. As informações são alimentadas baseadas na especificação para o processo de projeto e vice-versa.

Como a especificação é desenvolvida detalhadamente, sua compreensão dessa especificação aumenta. A criação de uma especificação formal força uma análise detalhada que geralmente revela erros e inconsistências na especificação informal de requisitos. Essa detecção de erros é provavelmente o argumento mais forte para se desenvolver uma especificação formal (Hall, 1990). Ela ajuda a descobrir problemas de requisitos cuja correção posterior pode ser muito onerosa.

Dependendo do processo usado, os problemas de especificação descobertos durante a análise formal podem causar mudanças na especificação de requisitos caso ela ainda não tenha sido aprovada. Se a especificação de requisitos tiver sido aprovada e incluída no contrato de desenvolvimento do sistema, você deverá apresentar os problemas encontrados ao cliente. Cabe, então, a ele decidir como os problemas devem ser resolvidos antes que você inicie o processo de projeto do sistema.

O desenvolvimento e a análise de uma especificação formal carregam a parte inicial dos custos de desenvolvimento de software. A Figura 10.3 mostra como provavelmente os custos de processo de software são afetados pelo uso da especificação formal. Quando um processo convencional é usado, os custos de validação constituem cerca de 50% dos custos de desenvolvimento, e os custos de projeto e implementação constituem cerca de duas vezes os custos de especificação. Com a especificação formal, os custos de especificação e implementação são comparáveis e os custos de validação são significativamente reduzidos. Como o desenvolvimento da especificação formal detecta os problemas de requisitos, evita-se o retrabalho para corrigir esses problemas após o projeto do sistema.

Duas abordagens fundamentais para especificação formal foram usadas para escrever especificações detalhadas para sistemas de software industriais. São elas:

Figura 10.1

Especificação e projeto.

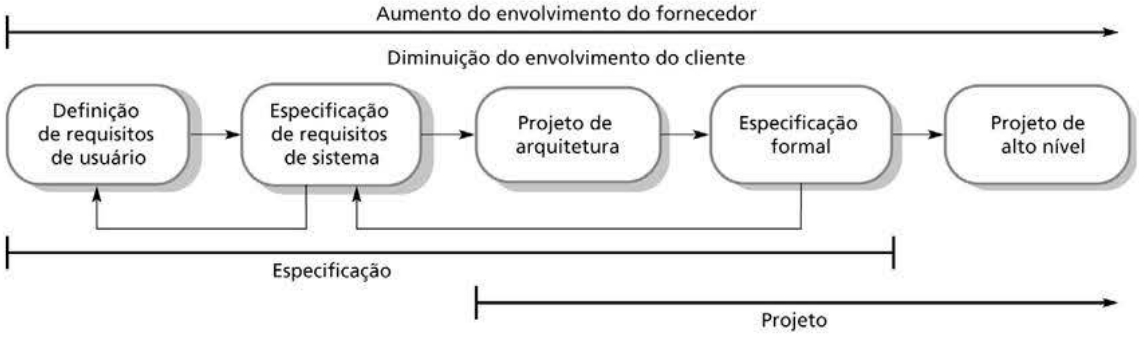
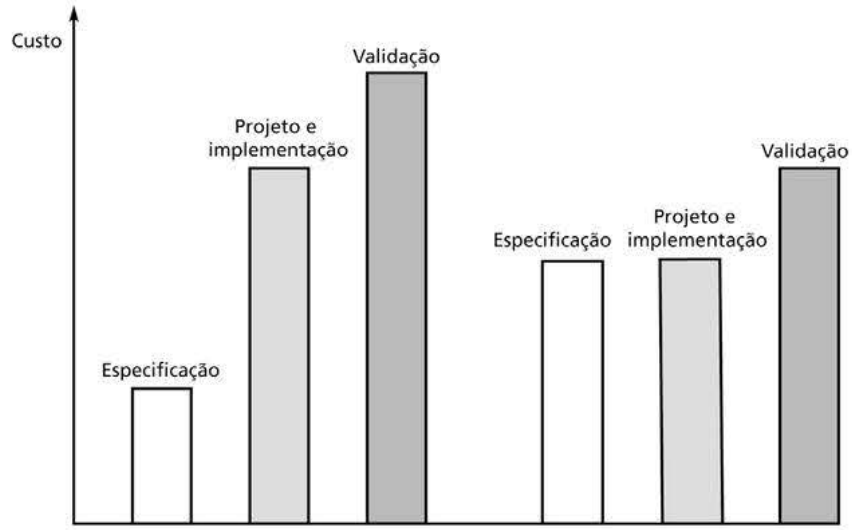


Figura 10.2

Especificação formal no processo de software.

Figura 10.3
Custos de desenvolvimento de software com especificação formal.



1. *Uma abordagem algébrica* na qual o sistema é descrito em termos de operações e seus relacionamentos.
2. *Uma abordagem baseada em modelos* na qual um modelo do sistema é criado por meio de construções matemáticas, como conjuntos e seqüências, e as operações de sistema são definidas em função de como elas modificam seu estado.

Diferentes linguagens dessas famílias foram desenvolvidas para especificar sistemas seqüenciais e concorrentes. A Tabela 10.1 mostra exemplos de linguagens em cada uma das classes. Você pode observar na tabela que a maioria das linguagens foi desenvolvida na década de 1980. São necessários muitos anos para refinar uma linguagem formal de especificação e, portanto, a maioria das pesquisas em especificação formal baseia-se atualmente nessas linguagens, e não existe interesse em definir novas notações.

Neste capítulo, a finalidade é apresentar ambas as abordagens, algébrica e baseada em modelos. Os exemplos devem fornecer uma idéia de como uma especificação formal resulta em uma especificação precisa e detalhada, mas não explico detalhadamente a linguagem de especificação, técnicas de especificação ou métodos de verificação do programa.

10.2 Especificação de interface de subsistema

Os sistemas de grande porte são em geral decompostos em subsistemas desenvolvidos independentemente. Os subsistemas fazem uso de outros subsistemas e, assim, uma parte essencial do processo de especificação é definir as interfaces de subsistemas. Uma vez que as interfaces tenham sido aprovadas e definidas, os subsistemas podem, depois, ser projetados e implementados independentemente.

As interfaces de subsistemas são com freqüência definidas como um conjunto de objetos ou componentes (Figura 10.4). Elas descrevem os dados e as operações que podem ser acessados por meio da interface de subsistema. Você pode, portanto, definir uma especificação de interface de subsistema combinando as especificações dos objetos que constituem a interface.

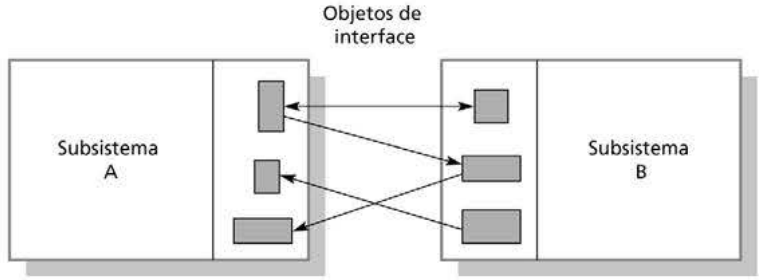
As especificações precisas de interface de subsistema são importantes, pois os desenvolvedores de subsistemas devem escrever códigos que usem os serviços de outros subsistemas antes de estes terem sido implementados. A especificação de interface fornece informações aos desenvolvedores de subsistemas de modo que eles conheçam quais serviços estarão disponíveis em outros subsistemas e como eles podem ser acessados. As especificações de interface de subsistema claras

Tabela 10.1 Linguagens formais de especificação

	Seqüencial	Concorrente
Algébrica	Larch (Guttag, et al., 1993); OBJ (Futatsugi, et al., 1985)	Lotos (Bolognesi e Brinksma, 1987)
Baseada em modelos	Z (Spivey, 1992); VDM (Jones, 1980); B (Wordsworth, 1996)	CSP (Hoare, 1985); redes de Petri (Peterson, 1981)

Figura 10.4

Objetos de interface de subsistemas.



e não ambíguas reduzem as chances de mal-entendidos entre um subsistema que fornece algum serviço e os subsistemas que usam o serviço.

A abordagem algébrica foi originalmente projetada para a definição de interfaces de tipo abstrato de dados. Em um tipo abstrato de dados, o tipo é definido por meio da especificação de operações de tipo, em vez da representação de tipo. Portanto, ele é similar a uma classe de objeto. O método algébrico da especificação formal define o tipo abstrato de dados em termos dos relacionamentos entre as operações de tipo.

Guttag (Guttag, 1977) foi o primeiro a explicar essa abordagem na especificação de tipos abstratos de dados. Cohen et al. (Cohen, et al., 1986) mostraram como a técnica pode ser estendida para especificação completa do sistema, usando o exemplo de um sistema de recuperação de documentos. Liskov e Guttag (Liskov e Guttag, 1986) também abordam a especificação de tipos abstratos de dados.

A estrutura de uma especificação de objeto é mostrada na Figura 10.5. O corpo da especificação tem quatro componentes:

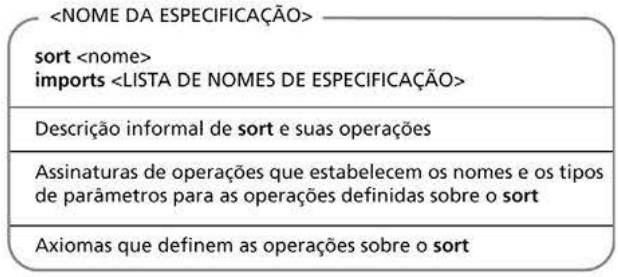
1. *Uma introdução* que declara o **sort** (nome do tipo) da entidade que está sendo especificada. **Sort** é o nome de um conjunto de objetos com características comuns. Ele é similar a um tipo em uma linguagem de programação. A introdução pode também incluir uma declaração de '**imports**', na qual os nomes de especificações que definem outros **sorts** são declarados. A importação de uma especificação torna esses **sorts** disponíveis para uso.
2. *Uma parte de descrição*, em que as operações são descritas informalmente. Ela facilita o entendimento da especificação formal. A especificação formal complementa essa descrição, fornecendo sintaxe e semântica não ambíguas para as operações de tipo.
3. *A parte de assinatura* define a sintaxe da interface da classe de objeto ou do tipo abstrato de dados. Na assinatura são descritos os nomes das operações definidas, o número e os tipos de seus parâmetros bem como o **sort** dos resultados de operações.
4. *A parte de axiomas* define a semântica das operações pela definição de um conjunto de *axiomas* que caracterizam o comportamento do tipo abstrato de dados. Esses axiomas relacionam as operações usadas para construir as entidades do **sort** definido com as operações usadas para inspecionar seus valores.

O processo de desenvolvimento de uma especificação formal de uma interface de subsistema inclui as seguintes atividades:

1. *Estruturação da especificação.* Organizar a especificação informal de interface em um conjunto de tipos abstratos de dados ou classes de objeto. Você deve definir informalmente as operações associadas a cada classe.
2. *Definição do nome da especificação.* Definir um nome para cada especificação de tipo abstrato de dados, decidir se eles requerem parâmetros genéricos e decidir os nomes para os **sorts** identificados.
3. *Seleção de operações.* Escolher um conjunto de operações para cada especificação com base na funcionalidade da interface identificada. Você deve incluir operações para criar instâncias do **sort**, modificar os valores das instâncias

Figura 10.5

Estrutura de uma especificação algébrica.



e inspecionar os valores das instâncias. Você pode precisar adicionar funções àquelas inicialmente identificadas na definição informal de interface.

4. *Especificação informal de operações.* Escrever uma especificação informal para cada operação. Você deve descrever como as operações afetam o **sort** definido.
5. *Definição de sintaxe.* Definir a sintaxe das operações e os parâmetros para cada uma delas. Esta é a parte de assinatura da especificação formal. Você deve atualizar a especificação informal neste estágio, se necessário.
6. *Definição de axiomas.* Definir a semântica das operações pela descrição de quais condições são sempre verdadeiras para as diferentes combinações de operações.

Para explicar a técnica de especificação algébrica, uso um exemplo de uma estrutura de dados simples (uma lista vinculada), conforme apresentada na Figura 10.6. Listas vinculadas são estruturas de dados ordenados nas quais cada elemento inclui uma ligação para o próximo elemento na estrutura. Usei uma lista simples, com apenas poucas operações associadas para que a explicação não ficasse muito longa. Na prática, as classes de objeto que definem uma lista teriam provavelmente mais operações.

Suponha que o primeiro estágio do processo de especificação, denominado estruturação de especificação, foi realizado e que a necessidade de uma lista foi identificada. O nome da especificação e o nome do **sort** podem ser os mesmos, embora seja útil distingui-los usando alguma convenção. Uso letras maiúsculas para o nome de especificação (**LIST**) e letras minúsculas com a inicial em maiúscula para o nome de **sort** (**List**). As listas são conjuntos de outros tipos e a especificação tem um parâmetro genérico (**Elem**). O nome **Elem** pode representar qualquer tipo: **integer**, **string**, **list** etc.

Em geral, para cada tipo abstrato de dados, as operações necessárias devem incluir uma operação para gerar instâncias do tipo (**Create**) e construir o tipo com base em seus elementos básicos (**Cons**). No caso de listas, deve existir uma operação para avaliar o primeiro elemento da lista (**Head**), uma operação que retorna a lista criada pela remoção do primeiro elemento (**Tail**) e uma operação para contar o número de elementos da lista (**Length**).

Para definir a sintaxe de cada uma dessas operações, você deve decidir quais parâmetros são necessários para a operação e os resultados desta. Em geral, os parâmetros de entrada são tanto o **sort** definido (**List**) quanto o **sort** genérico (**Elem**). Os resultados das operações podem ser tanto esses **sorts** ou algum outro **sort** como **Integer** ou **Boolean**. No exemplo da lista, a operação **Length** retorna um inteiro. Portanto, você deve incluir uma declaração '**imports**' para declarar que a especificação de inteiro é usada na especificação.

Para criar a especificação, defina um conjunto de axiomas que se aplicam ao tipo abstrato de dados e especificam sua semântica. Os axiomas são definidos por meio das operações definidas na parte de assinatura. Esses axiomas especificam a semântica pela definição do que é sempre verdadeiro em relação ao comportamento das entidades com o tipo abstrato de dados.

As operações sobre um tipo abstrato de dados geralmente se dividem em duas categorias:

1. *Operações de construtor* que criam ou modificam as entidades do **sort** definido na especificação. Em geral recebem nomes como **Create**, **Update**, **Add** ou, neste caso, **Cons**, que significa construir.
2. *Operações de inspeção* que avaliam os atributos do **sort** definidos na especificação. Geralmente recebem nomes como **Eval** ou **Get**.

Figura 10.6

Simples especificação de lista.

```
LIST ( Elem )
sort List
imports INTEGER

Define uma lista na qual os elementos são adicionados no fim e removidos
do início. As operações são Create, que cria uma lista vazia, Cons, que cria
uma nova lista com um membro adicionado, Length, que avalia o tamanho
da lista, Head, que avalia o primeiro elemento da lista, e Tail, que cria uma
lista ao remover o primeiro elemento de sua lista de entrada. Undefined
representa um valor indefinido do tipo Elem.

Create → List
Cons (List, Elem) → List
Head (List) → Elem
Length (List) → Integer
Tail (List) → List

Head (Create) = Undefined exception (empty list)
Head (Cons (L, v)) = if L = Create then v else Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = if L = Create then Create else Cons (Tail (L), v)
```

Uma boa regra prática para escrever uma especificação algébrica é estabelecer as operações de construtor e escrever um axioma para cada operação de inspeção sobre cada construtor. Isso sugere que, se existirem **m** operações de construtor e **n** operações de inspeção, devem existir **m . n** axiomas definidos.

No entanto, pode ser que nem todas as operações de construtor associadas a um tipo abstrato de dados sejam primitivas. Isto é, pode ser possível defini-las usando outras operações de construtores e de inspeção. Se uma operação de construtor for definida usando outros construtores, você vai precisar definir apenas as operações de inspeção usando os construtores primitivos.

Na especificação de lista, as operações de construtor que constroem as listas são **Create**, **Cons** e **Tail**. As operações de inspeção são **Head** (retorna o valor do primeiro elemento na lista) e **Length** (retorna o número de elementos na lista), as quais são usadas para descobrir os atributos da lista. A operação **Tail**, no entanto, não é um construtor primitivo. Portanto, não é necessário definir axiomas sobre a operação **Tail** para as operações **Head** e **Length**, mas você deve definir **Tail** usando as operações primitivas de construtor.

A avaliação do primeiro elemento de uma lista vazia resulta em um valor indefinido. As especificações de **Head** e **Tail** mostram que **Head** avalia o início da lista e **Tail** avalia a lista de entrada com o seu primeiro elemento removido. A especificação de **Head** estabelece que o primeiro elemento de uma lista criado por meio de **Cons** é o valor adicionado à lista (se a lista inicial estiver vazia) ou é igual ao primeiro elemento do parâmetro inicial de lista para **Cons**. A adição de um elemento em uma lista não afeta seu primeiro elemento a menos que a lista esteja vazia.

A recursão é comumente usada quando são escritas especificações algébricas. O valor da operação **Tail** é a lista formada com base na lista de entrada e com a remoção do primeiro elemento. A definição de **Tail** mostra como a recursão é usada na construção de especificações algébricas. A operação é definida sobre listas vazias e, depois, recursivamente, sobre listas não vazias, sendo que a recursão termina quando o resultado for uma lista vazia.

Às vezes é mais fácil compreender as especificações recursivas com o desenvolvimento de um breve exemplo. Digamos que haja uma lista [**5, 7**] na qual **5** é o início da lista e **7**, o fim da lista. A operação **Cons** (**[5, 7], 9**) deve retornar uma lista **[5, 7, 9]**, e a operação **Tail** aplicada a ela deve retornar a lista [**7, 9**]. A seqüência de equações que resulta da substituição dos parâmetros na especificação anterior com esses valores é:

Tail ([5, 7, 9]) =
Tail (Cons ([5, 7], 9)) = Cons (Tail ([5, 7]), 9) =
Cons (Tail (Cons ([5], 7)), 9) = Cons (Cons (Tail ([5]), 7), 9) =
Cons (Cons (Tail (Cons ([], 5)), 7), 9) = Cons (Cons ([Create], 7), 9) =
Cons ([7], 9) = [7, 9]

A reescrita sistemática do axioma de **Tail** ilustra que este realmente produz o resultado previsto. Você pode verificar que o axioma para **Head** está correto usando a mesma técnica de reescrita.

Agora, vejamos como você pode usar a especificação algébrica de uma interface em uma especificação de sistema crítico. Suponha que, em um sistema de controle de tráfego aéreo, um objeto foi projetado para representar um setor controlado do espaço aéreo. Cada setor controlado pode conter uma série de aeronaves, cada um dos quais tem um único identificador de aeronave. Por motivos de segurança, todas as aeronaves devem estar separadas uma das outras por pelo menos 300 metros em altitude. O sistema avisará o controlador caso seja feita uma tentativa de posicionar uma aeronave de modo que essa restrição seja desrespeitada.

Para simplificar a descrição, defini somente um número limitado de operações no objeto **sector**. Em um sistema real existem provavelmente muito mais operações e condições mais complexas de segurança relacionadas à separação horizontal da aeronave. As operações críticas sobre o objeto são:

1. **Enter.** Essa operação adiciona uma aeronave (representada por um identificador) ao espaço aéreo em uma altura especificada. Não deve haver outra aeronave nessa altura ou a 300 metros dela.
2. **Leave.** Essa operação remove a aeronave especificada do setor controlado. Tal operação é usada quando a aeronave se move para um setor adjacente.
3. **Move.** Essa operação move uma aeronave de uma altitude para outra. Novamente, a restrição de segurança de que a separação vertical da aeronave deve ser de, pelo menos, 300 metros é verificada.
4. **Lookup.** Dado um identificador da aeronave, essa operação retorna a altitude real da aeronave no setor.

Isso torna mais fácil especificar essas operações se algumas outras operações forem definidas. São elas:

1. **Create.** É uma operação-padrão para um tipo abstrato de dados. Causa a criação de uma instância vazia do tipo. Neste caso, representa um setor que não contém aeronave.
2. **Put.** É uma versão mais simples da operação **Enter**. Adiciona uma aeronave ao setor sem nenhuma verificação de restrição associada.

3. **In-space.** Dado um sinal de chamada de aeronave, essa operação booleana retornará verdadeiro se a aeronave estiver no setor controlado e falso em caso contrário.
4. **Occupied.** Dada uma altitude, essa operação booleana retornará verdadeiro caso exista uma aeronave dentro de 300 metros dessa altitude e falso em caso contrário.

A vantagem de definir essas operações simples é que você pode usá-las como *building blocks* para definir operações mais complexas sobre o **sort Sector**. A especificação algébrica desse **sort** é mostrada na Figura 10.7.

Essencialmente, as operações básicas de construtor são **Create** e **Put**, e uso essas operações na especificação de outras operações. **Occupied** e **In-space** são operações de verificação que defini usando **Create** e **Put** e, depois, usei-as em outras especificações. Não há como explicar todas as operações detalhadamente nesta seção, mas explico duas delas (**Occupied** e **Move**). Com essas informações você deve ser capaz de compreender as outras especificações da operação.

1. A operação **Occupied** toma um setor e um parâmetro que representa a altitude e verifica se uma aeronave foi atribuída a essa altitude. Sua especificação estabelece que:

Figura 10.7

Especificação de um setor controlado.

```
SECTOR
sort Sector
imports INTEGER, BOOLEAN

Enter     — adiciona uma aeronave no setor se as condições de segurança forem satisfeitas
Leave     — remove uma aeronave do setor
Move      — move uma aeronave de uma altitude para outra, se for seguro
Lookup    — encontra a altitude de uma aeronave no setor

Create    — cria um setor vazio
Put       — adiciona uma aeronave a um setor sem verificações de restrições
In-space  — verifica se uma aeronave já está em um setor
Occupied  — verifica se uma altitude especificada está disponível

Enter (Sector, Call-sign, Height) → Sector
Leave (Sector, Call-sign) → Sector
Move (Sector, Call-sign, Height) → Sector
 Lookup (Sector, Call-sign) → Height

Create → Sector
Put (Sector, Call-sign, Height) → Sector
In-space (Sector, Call-sign) → Boolean
Occupied (Sector, Height) → Boolean

Enter (S, CS, H) =
   if      In-space (S, CS) then S exception (Aircraft already in sector)
   elsif   Occupied (S, H) then S exception (Height conflict)
   else    Put (S, CS, H)

Leave (Create, CS) = Create exception (Aircraft not in sector)
Leave (Put (S, CS1, H1), CS) =
   if CS = CS1 then S else Put (Leave (S, CS), CS1 , H1)

Move (S, CS, H) =
   if      S = Create then Create exception (No aircraft in sector)
   elsif   not In-space (S, CS) then S exception (Aircraft not in sector)
   elsif   Occupied (S, H) then S exception (Height conflict)
   else    Put (Leave (S, CS), CS, H)

-- NO-HEIGTH é uma constante que indica se uma altitude válida não pode ser retornada

Lookup (Create, CS) = NO-HEIGHT exception (Aircraft not in sector)
Lookup (Put (S, CS1, H1), CS) =
   if CS = CS1 then H1 else Lookup (S, CS)

Occupied (Create, H) = false
Occupied (Put (S, CS1 , H1), H) =
   if      (H1 > H and H1 - H ≤ 300) or (H > H1 and H - H1 ≤ 300) then true
   else    Occupied (S, H)

In-space (Create, CS) = false
In-space (Put (S, CS1, H1), CS) =
   if CS = CS1 then true else In-space (S, CS)
```

- Em um setor vazio (aquele criado por uma operação **Create**), todos os níveis estão livres. A operação retorna falso, independentemente do valor do parâmetro de altitude.
- Em um setor não vazio (onde houve operações **Put** anteriores), a operação **Occupied** verificará se a altitude especificada (parâmetro **H**) está dentro dos 300 metros da altitude da aeronave adicionada por último ao setor pela operação **Put**. Se sim, essa altitude já está ocupada, portanto o valor de **Occupied** será verdadeiro.
- Se não estiver ocupada, a operação verificará o setor recursivamente. Você pode pensar nessa verificação como se fosse realizada sobre a última aeronave inserida no setor. Se a altitude não estiver dentro da faixa de altitude dessa aeronave, a operação executará a verificação em relação à aeronave anterior inserida, e assim por diante. Eventualmente, caso não exista uma aeronave na faixa de altitude especificada, a verificação será realizada em relação a um setor vazio e, então, retornará falso.

2. A operação **Move** move uma aeronave de uma altitude para outra em um setor. Sua especificação estabelece que:
   - Se uma operação **Move** for aplicada a um espaço aéreo vazio (resultado de **Create**), o espaço aéreo não será alterado e uma exceção será lançada para indicar que a aeronave especificada não está no espaço aéreo.
   - Em um setor não vazio, a operação verificará inicialmente (usando **In-space**) se a aeronave determinada está no setor. Se não estiver, uma exceção será lançada. Se estiver, a operação verificará se a altitude especificada está disponível (usando **Occupied**), lançando uma exceção caso já exista uma aeronave nessa altitude.
   - Se a altitude especificada estiver disponível, a operação **Move** será equivalente a aeronave especificada deixar o espaço aéreo (então, a operação **Leave** é usada) e ser colocada dentro do setor na nova altitude.

10.3 Especificação de comportamento

As técnicas algébricas simples descritas na seção anterior podem ser usadas para descrever interfaces em que as operações de objeto são independentes do estado do objeto. Isto é, os resultados da aplicação de uma operação não devem depender dos resultados das operações anteriores. Onde essa condição não prevalece, as técnicas algébricas podem se tornar incômodas. Além disso, como elas aumentam de tamanho, considero que as descrições algébricas de comportamento de sistema tornam-se mais difíceis de serem compreendidas.

Uma abordagem alternativa para a especificação formal mais amplamente usada em projetos industriais é a especificação baseada em modelos. Ela é uma abordagem para especificação formal na qual a especificação do sistema é expressa como um modelo de estados de sistema. Você pode especificar as operações de sistema pela definição de como elas afetam o estado do modelo de sistema. A combinação dessas especificações define o comportamento geral do sistema.

As notações maduras para desenvolvimento de especificações baseadas em modelos são VDM (Jones, 1980; Jones, 1986), B (Wordsworth, 1996) e Z (Hayes, 1987; Spivey, 1992). Uso Z (pronuncia-se 'Zed', e não 'Zee') nesta seção. Em Z, os sistemas são modelados com a utilização de conjuntos e relações entre conjuntos. Entretanto, Z expandiu esses conceitos matemáticos com construções que apóiam a especificação de software em especial.

Em uma introdução à especificação baseada em modelos, posso fornecer somente uma visão geral de como uma especificação pode ser desenvolvida. Uma descrição completa da notação Z seria mais extensa do que a apresentada neste capítulo. Em vez disso, apresento alguns pequenos exemplos para ilustrar a técnica e apresentar a notação à medida que ela for necessária. Uma descrição completa da notação Z é dada em livros como o de Diller (Potter, et al., 1996) e Jacky (Jacky, 1997).

As especificações formais podem ser difíceis e tediosas de ler, especialmente quando são apresentadas como grandes fórmulas matemáticas. Os projetistas de Z tomaram cuidado especial com esse problema. As especificações são apresentadas como texto informal, complementado com descrições formais. A descrição formal é incluída como pequenos trechos fáceis de ler (denominados *esquemas*) distintos do texto associado por meio de destaques gráficos. Os esquemas são usados para introduzir as variáveis de estado e para definir restrições e operações sobre o estado. Os esquemas podem, por si sós, ser manipulados com o uso de operações como composição, renomeação e ocultação de esquemas.

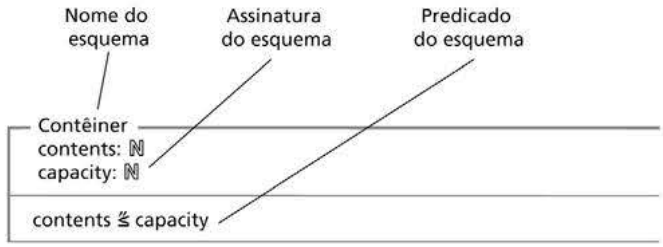
Para ser mais eficiente, uma especificação formal deve ser complementada por uma descrição informal de apoio. A apresentação do esquema Z foi projetada de modo que ele se destaque do texto ao redor (Figura 10.8).

A assinatura do esquema define as entidades que constituem o estado do sistema, e o predicado do esquema estabelece as condições que devem ser sempre verdadeiras para essas entidades. Quando um esquema define uma operação, o predicado pode estabelecer as pré e pós-condições. Elas definem o estado antes e depois da operação. A diferença entre essas pré e pós-condições define a ação especificada no esquema da operação.

Para ilustrar o uso de Z na especificação de um sistema crítico, foi desenvolvida uma especificação formal do sistema de controle da bomba de insulina que apresentei no Capítulo 3.

Figura 10.8

Estrutura de um esquema Z.



Recordando, esse sistema monitora o nível de glicose no sangue de diabéticos e injeta automaticamente a insulina, se necessário. Mesmo para um sistema pequeno, como a bomba de insulina, a especificação formal é bastante longa. Embora a operação básica do sistema seja simples, existem várias condições possíveis de alerta que devem ser consideradas. Incluí somente alguns dos esquemas que definem o sistema; a especificação completa pode ser baixada do site Web do livro.

Para desenvolver uma especificação baseada em modelos, você precisa definir as variáveis de estado e os predicados que modelam o estado do sistema que está sendo especificado, assim como as invariantes (condições que são sempre verdadeiras) sobre essas variáveis de estado.

O esquema de estados de Z que modela o estado da bomba de insulina está apresentado na Figura 10.9. Você pode observar como as duas partes básicas são usadas. Na parte superior, nomes e tipos são declarados e, na parte inferior, as invariantes são declaradas.

Figura 10.9

Esquema State para a bomba de insulina.

```
INSULIN_PUMP_STATE
  //Definição de dispositivo de entrada
  switch?: (off, manual, auto)
  ManualDeliveryButton?: ℕ
  Reading?: ℕ
  HardwareTest?: (OK, batterylow, pumpfail, sensorfail, deliveryfail)
  InsulinReservoir?: (present, notpresent)
  Needle?: (present, notpresent)
  clock?: TIME

  //Definição de dispositivo de saída
  alarm! = (on, off)
  display1!: string
  display2!: string
  clock!: TIME
  dose!: ℕ

  //Variáveis de estado usadas para o cálculo da dose
  status: (running, warning, error)
  r0, r1, r2: ℕ
  capacity, insulin_available : ℕ
  max_daily_dose, max_single_dose, minimum_dose: ℕ
  safemin, safemax: ℕ
  CompDose, cumulative_dose: ℕ
  ────────────────────────────────────────────────
  r2 = Reading?
  dose! ≤ insulin_available
  insulin_available ≤ capacity

  // A dose cumulativa de insulina fornecida zera uma vez a cada 24 horas
  clock? = 000000 ⇒ cumulative_dose = 0

  // Se a dose cumulativa exceder o limite, então a operação será suspensa
   cumulative_dose ≥ max_daily_dose ∧ status = error ⇒
       display1! = "Daily dose exceeded"

  // Parâmetros de configuração da bomba
  capacity = 100 ∧ safemin = 6 ∧ safemax = 14
  max_daily_dose = 25 ∧ max_single_dose = 4 ∧ minimum_dose = 1

  display2! = nat_to_string (dose!)
  clock! = clock?
```

Os nomes declarados no esquema são usados para representar as entradas de sistema, saídas de sistema e variáveis de estado internas:

1. *Entradas de sistema* para as quais a convenção em Z dita que todos os nomes de variáveis de entrada devem ser seguidos pelo símbolo !. Declarei nomes para modelar a chave liga/desliga da bomba (**switch?**), um botão para fornecimento manual de insulina (**ManualDeliveryButton?**), a leitura obtida do sensor de açúcar no sangue (**Reading?**), o resultado da execução de um programa de teste de hardware (**HardwareTest?**), sensores que detectam a presença do reservatório de insulina e da agulha (**InsulinReservoir?, Needle?**) e o valor do horário atual (**clock?**).
2. *Saídas de sistema* para as quais a convenção em Z dita que todos os nomes de variável de saída devem ser seguidos pelo símbolo !. Declarei nomes para modelar o alarme da bomba (**alarm!**), dois displays alfanuméricos (**display1!** e **display2!**), um display de horário atual (**clock!**) e a dose de insulina a ser fornecida (**dose!**).
3. *Variáveis de estado usadas para o cálculo de dose*. Declarei variáveis para representar o status do dispositivo (**status**), para manter os valores anteriores do nível de açúcar no sangue (**r0, r1 e r2**), a capacidade do reservatório de insulina e a quantidade de insulina disponível atualmente (**capacity, insulin_available**), diversas variáveis usadas para impor limites à dose de insulina fornecida (**max_daily_dose, max_single_dose, minimum_dose, safemin, safemax**) e duas variáveis usadas no cálculo de dose (**CompDose** e **cumulative_dose**). O tipo $\mathbb{N}$ significa um número não negativo.

O predicado do esquema define as invariantes que são sempre verdadeiras. Existe um 'e' implícito entre cada linha de predicado, de modo que todos os predicados devem se manter o tempo todo. Alguns desses predicados simplesmente estabelecem limites ao sistema, mas outros definem as condições fundamentais de operação do sistema. Estas incluem:

1. A dose deve ser menor ou igual à capacidade do reservatório de insulina. Isto é, é impossível fornecer mais insulina do que existe no reservatório.
2. A dose cumulativa é reiniciada à meia-noite de cada dia. Você pode pensar na frase de Z <expressão lógica 1> ⇒ <expressão lógica 2> como se fosse a mesma que **if** <expressão lógica 1> **then** <expressão lógica 2>. Neste caso, <expressão lógica 1> é '**clock? = 000000**' e <expressão lógica 2> é '**cumulative_dose = 0**'.
3. A dose cumulativa fornecida ao longo de um período de 24 horas não pode exceder **max_daily_dose**. Se essa condição for falsa, será emitida uma mensagem de erro.
4. **display2!** sempre mostra o valor da última dose de insulina fornecida e **clock!** sempre mostra o horário atual do relógio.

A bomba de insulina opera verificando a glicose do sangue a cada 10 minutos, e (de forma simples) a insulina será fornecida se a taxa de mudança de glicose do sangue estiver aumentando. O esquema **RUN**, mostrado na Figura 10.10, modela a condição de operação normal da bomba.

Se um nome do esquema for incluído na parte de declarações, isso equivale a incluir todos os nomes declarados nesse esquema na parte de declarações e as condições na parte de predicados. O esquema delta (Δ) na primeira linha da Figura 10.10 ilustra isso. O delta significa que as variáveis de estado definidas em **INSULIN_PUMP_STATE** estão no escopo, assim como está o conjunto de outras variáveis que representam os valores de estado antes e depois de alguma operação. Elas são indicadas pelo destaque do nome definido em **INSULIN_PUMP_STATE**. Portanto, **insulin_available** representa a quantidade de insulina disponível antes de alguma operação e **insulin_available** representa a quantidade de insulina disponível após alguma operação.

O esquema **RUN** define as operações do sistema pela especificação de um conjunto de predicados verdadeiros no uso normal de sistema. Naturalmente, estes adicionam-se aos predicados definidos no esquema **INSULIN_PUMP_STATE** que são invariantes (sempre verdadeiras). Esse esquema também mostra o uso de um recurso de Z — composição de esquema — no qual os esquemas **SUGAR_LOW**, **SUGAR_OK** e **SUGAR_HIGH** são incluídos ao se fornecer seus nomes. Observe que esses esquemas estão ligados pela operação 'ou', de modo que existe um esquema para cada uma das possíveis condições. A capacidade de compor os esquemas significa que você pode quebrar uma especificação em partes menores, da mesma maneira que pode definir funções e métodos em um programa.

O esquema **RUN** não será descrito em detalhes, mas, essencialmente, ele começa definindo os predicados verdadeiros para operação normal. Por exemplo, ele declara que a operação normal será possível apenas quando a quantidade de insulina disponível for maior do que a dose única máxima que pode ser fornecida. Os três esquemas que representam diferentes níveis de açúcar no sangue são ligados pela operação 'ou' e, conforme veremos mais adiante, definem o valor para a variável de estado **CompDose**.

O valor de **CompDose** representa a quantidade de insulina calculada para ser fornecida com base no nível de açúcar no sangue. O restante dos predicados nesse esquema define várias verificações a serem aplicadas para assegurar que a dose realmente fornecida (**dose!**) segue as regras de segurança definidas para o sistema. Por exemplo, uma regra de segurança é que nenhuma dose única de insulina pode exceder nenhum valor máximo definido.

Figura 10.10

Esquema RUN.

```
RUN
INSULIN_PUMP_STATE
switch? = auto _
status = running ∨ status = warning
insulin_available ≥ max_single_dose
cumulative_dose < max_daily_dose
// A dose de insulina é calculada dependendo do nível de açúcar no sangue
(SUGAR_LOW ∨ SUGAR_OK ∨ SUGAR_HIGH)
// 1. Se a dose de insulina calculada for zero, não fornecer insulina
CompDose = 0 ⇒ dose! = 0
∨
// 2. A dose máxima diária seria excedida se a dose calculada fosse fornecida, de
   modo que a dose de insulina é estabelecida pela diferença entre a dose máxima
   diária permitida e a dose cumulativa fornecida até o momento presente
CompDose + cumulative_dose > max_daily_dose ⇒ alarm! = on ∧ status' =
   warning ∧ dose! = max_daily_dose–cumulative_dose
∨
// 3. Situação normal. Se a dose única máxima não for excedida, então fornecer
   a dose calculada. Se a dose única calculada for muito alta, restringir a dose
   fornecida para a dose única máxima
CompDose + cumulative_dose < max_daily_dose ⇒
        ( CompDose ≤ max_single_dose ⇒ dose! = CompDose
        ∨
        CompDose > max_single_dose ⇒ dose! = max_single_dose )
insulin_available' = insulin_available–dose!
cumulative_dose' = cumulative_dose + dose!

insulin_available ≤ max_single_dose * 4 ⇒ status' = warning ∧
display1! = "Insulin low"

r1' = r2
r0' = r1
```

Finalmente, os dois últimos predicados definem as mudanças de valor de **insulin_available** e **cumulative_dose**. Observe como usei a versão destacada dos nomes aqui.

O exemplo final do esquema apresentado na Figura 10.11 define como a dose de insulina é calculada, supondo que o nível de açúcar no sangue do diabético esteja dentro de uma área segura. Nessas circunstâncias, a insulina será fornecida somente se o nível de açúcar no sangue estiver subindo e a taxa de mudança do nível de açúcar no sangue estiver aumentando. Os outros esquemas, **SUGAR_LOW** e **SUGAR_HIGH**, definem a dose a ser fornecida se o nível de açúcar estiver fora da área segura. Os predicados no esquema são os seguintes:

1. O predicado inicial define a área segura; isto é, **r2** deve estar entre **safemin** e **safemax**.
2. Se o nível de açúcar estiver estável ou caindo, indicado por **r2** (a leitura mais recente) igual ou menor que **r1** (leitura inicial), a dose da insulina a ser fornecida será zero.

Figura 10.11

O esquema SUGAR_OK.

```
SUGAR_OK
r2 ≥ safemin ∨ r2 ∧ safemax
// nível de açúcar estável ou decrescente
r2 ≤ r1 ⇒ CompDose = 0
∨
// nível de açúcar crescente, mas com taxa de aumento decrescente
r2 > r1 ∧ (r2-r1) < (r1-r0) ⇒ CompDose = 0
∨
// nível de açúcar crescente e taxa de aumento crescente, calcular dose
// uma dose mínima deve ser fornecida se próximo de zero
r2 > r1 ∧ (r2-r1) ≥ (r1-r0) ∧ (round ((r2-r1)/4) = 0) ⇒
    CompDose = minimum_dose
∨
r2 > r1 ∧ (r2-r1) ≥ (r1-r0) ∧ (round ((r2-r1)/4) > 0) ⇒
    CompDose = round ((r2-r1)/4)
```

3. Se o nível de açúcar estiver aumentando (**r2 > r1**), mas a taxa de aumento estiver decrescente, a dose a ser fornecida será zero.
4. Se o nível de açúcar estiver aumentando e a taxa de aumento estiver estável, uma dose mínima de insulina será fornecida.
5. Se o nível de açúcar estiver aumentando e a taxa de aumento estiver crescendo, a dose de insulina a ser fornecida será derivada aplicando-se uma fórmula simples aos valores calculados.

Não modelo o comportamento temporal do sistema (isto é, o fato de que o sensor de glicose seja verificado a cada 10 minutos) usando Z. Embora isso seja realmente possível, é um tanto inadequado e, em meu ponto de vista, uma descrição informal comunica a especificação mais concisamente do que uma especificação formal.

PONTOS-CHAVE

- Os métodos de especificação formal de sistemas complementam as técnicas de especificação informal de requisitos. Eles podem ser usados com uma definição de requisitos em linguagem natural para esclarecer algumas áreas de potencial ambigüidade na especificação.
- Especificações formais são precisas e não ambíguas. Elas removem áreas duvidosas em uma especificação e evitam alguns dos problemas de má interpretação de linguagem. Contudo, os não-especialistas podem achar as especificações formais difíceis de serem compreendidas.
- A principal vantagem do uso de métodos formais no processo de software é que eles forçam uma análise de requisitos de sistema no início do estágio. A correção de erros nesse estágio é mais barata que a modificação de um sistema já entregue.
- As técnicas de especificação formal são mais adequadas quanto ao custo no desenvolvimento de sistemas críticos nos quais segurança, confiabilidade e proteção são particularmente importantes. Elas podem ser também usadas para especificar padrões.
- As técnicas algébricas de especificação formal são particularmente adequadas para a especificação de interfaces nas quais a interface é definida como um conjunto de classes de objeto ou tipos abstratos de dados. Essas técnicas ocultam o estado do sistema e especificam o sistema em termos de relacionamentos entre as operações de interface.
- As técnicas baseadas em modelos modelam o sistema usando construções matemáticas como conjuntos e funções. Elas podem expor o estado do sistema, o que simplifica alguns tipos de especificação de comportamento.
- As operações em uma especificação baseada em modelos são definidas por pré e pós-condições sobre o estado do sistema.

LEITURAS SUGERIDAS

"Correctness by construction: developing a commercially secure system". Uma boa descrição de como os métodos formais podem ser usados no desenvolvimento de um sistema crítico de proteção. (A. Hall e R. Chapman, *IEEE software*, 19(1), janeiro de 2002.)

IEEE transactions on software engineering, janeiro de 1998. Esta edição da revista inclui uma seção especial sobre os usos práticos dos métodos formais em engenharia de software. Ela inclui artigos sobre Z e LARCH.

"Formal methods: promises and problems". Este artigo é uma explicação realista dos ganhos potenciais baseados em métodos formais e nas dificuldades de integração do uso de métodos formais no desenvolvimento prático de software. (Luqi e J. Goguen. *IEEE software*, 14(1), janeiro de 1997.)

EXERCÍCIOS

10.1 Sugira por que o projeto de arquitetura de um sistema deve preceder o desenvolvimento de uma especificação formal.

10.2 Você recebeu a tarefa de 'vender' técnicas formais de especificação para uma organização de desenvolvimento de software. Faça um esboço de como você explicaria as vantagens de especificações formais para engenheiros de software céticos e práticos.

10.3 Explique por que é particularmente importante definir interfaces de subsistemas de maneira precisa e por que a especificação algébrica é particularmente apropriada para a especificação da interface do subsistema.

10.4 Um tipo abstrato de dados que representa uma pilha tem as seguintes operações associadas a ela:

New: Gera uma nova pilha.

Push: Adiciona um elemento no topo da pilha.

Top: Avalia o elemento no topo da pilha.

Retract: Remove o elemento do topo e retorna a pilha modificada.

Empty: Verdadeiro, se não existirem elementos na pilha.

Defina esse tipo abstrato de dados usando uma especificação algébrica.

10.5 No exemplo de espaço aéreo controlado, a condição de segurança é que uma aeronave não pode estar dentro de 300 metros de altitude no mesmo setor. Modifique a especificação mostrada na Figura 10.6 para permitir que uma aeronave ocupe a mesma altitude no setor, desde que elas estejam separadas pelo menos por 8 km de diferença horizontal. Você pode ignorar aeronaves em setores adjacentes. *Dica*: você deve modificar as operações de construtor de modo que incluam a posição da aeronave bem como sua altitude. Também deve ser definida uma operação que, dadas duas posições, retorne a separação entre elas.

10.6 Os caixas eletrônicos de bancos contam com o uso de informações do cartão do usuário que fornece o identificador de banco, o número de conta e o identificador pessoal do usuário. Eles também obtêm informações da conta com base em um banco de dados central e atualizam esse banco de dados quando a transação é completada. Usando seu conhecimento de operação de caixa eletrônico, escreva os esquemas em Z definindo o estado do sistema, a validação de cartão (no qual o identificador do usuário é verificado) e a retirada de dinheiro.

10.7 Modifique o esquema da bomba de insulina, apresentado na Figura 10.8, para adicionar a seguinte condição de segurança em que **ManualDeliveryButton?** pode somente ter um valor diferente de zero caso a chave da bomba esteja na posição manual.

10.8 Escreva um esquema em Z chamado **SELF_TEST** para testar os componentes de hardware da bomba de insulina e estabelecer o valor da variável de estado **HardwareTest?**. Em seguida, modifique o esquema **RUN** para verificar se o hardware está sendo operado com sucesso antes de fornecer alguma insulina. Se não, a dose fornecida deverá ser zero e um erro deverá ser indicado no display da bomba de insulina.

10.9 Z apóia o conceito de seqüências, no qual uma seqüência é como um vetor. Por exemplo, para uma seqüência S, você pode se referir aos seus elementos como **S[1]**, **S[2]** e assim por diante. Ele também permite que você determine o número de elementos de uma seqüência usando o operador. Isto é, se uma seqüência **S** for **[a, b, c, d]**, então **S** será **4**. Você pode adicionar um elemento no fim de uma seqüência **S**, escrevendo S + a, e no início da seqüência escrevendo **a + S**. Usando essas construções, escreva uma especificação de **LIST** em Z especificada algebricamente na Figura 10.6.

10.10 Você é um engenheiro de sistema e lhe foi solicitado sugerir a melhor maneira de desenvolver o software crítico de segurança para um marca-passo. Você sugeriu a especificação formal do sistema, mas seu gerente recusou a sugestão. Você considera que as razões dele são fracas e preconceituosas. É ético desenvolver o sistema usando métodos que você considera inadequados?

PARTE 3 PROJETO

A essência do projeto de software é tomar decisões sobre a organização lógica do software. Algumas vezes, essa organização lógica é representada como um modelo em uma linguagem de modelagem definida, como a UML, e algumas vezes simplesmente se usam notações informais e rascunhos para representar o projeto. É claro que você raramente parte do zero quando toma decisões sobre a organização do software, baseando seu projeto em experiências anteriores.

Alguns autores pensam que a melhor maneira de englobar essa experiência é em métodos estruturados nos quais você segue um processo de projeto definido e descreve seu projeto usando tipos diferentes de modelo. Nunca fui um grande admirador de métodos estruturados, pois sempre considerei-os muito restritivos. O projeto é um processo criativo e acredito firmemente que cada um tem sua própria maneira de perceber esses processos criativos. Não há maneira certa ou errada de projetar software, nem eu ou qualquer outra pessoa poderá lhe dar uma 'receita' para projetar software. Você aprende como projetar estudando exemplos de projetos existentes e discutindo seu projeto com outros.

Mais do que representar experiências como um 'método de projeto', prefiro uma abordagem não tão estruturada. Os capítulos desta parte englobam conhecimento sobre estruturas de software usadas com sucesso em outros sistemas, apresentam alguns exemplos e dão alguns conselhos sobre processos de projeto:

Os Capítulos 11 a 13 tratam das estruturas abstratas de software. O Capítulo 11 explica perspectivas estruturais consideradas úteis no projeto de software, o Capítulo 12 trata da estruturação de software para execução distribuída e o Capítulo 13 trata de estruturas genéricas para vários tipos de aplicações. O Capítulo 13 é um capítulo novo que incluí nesta edição porque encontrei muitos estudantes de engenharia de software sem experiência em aplicações de software, apesar dos sistemas interativos que usam diariamente em seus próprios computadores.

Os Capítulos 14 a 16 estão relacionados a tópicos de projetos de software mais específicos. O Capítulo 14, que aborda o projeto orientado a objetos, trata de um método de raciocínio sobre estruturas de software. O Capítulo 15 aborda o projeto de sistemas de tempo real e explica as estruturas de software necessárias em sistemas em que a resposta em tempo adequado é um requisito crítico. O Capítulo 16 é um pouco diferente, pois enfoca o projeto de interface com o usuário em vez de estruturas de software. Como engenheiro, você precisa pensar em sistemas — não somente em software —, e as pessoas são um componente essencial no sistema. O projeto não pára nas estruturas de software, ele envolve a maneira como o software é usado.

11

Projeto de arquitetura

Objetivos

O objetivo deste capítulo é apresentar os conceitos de arquitetura de software e projeto de arquitetura. Após ler este capítulo, você:

- compreenderá por que o projeto de arquitetura de software é importante;
- saberá quais decisões devem ser tomadas sobre a arquitetura de sistema durante o processo de projeto de arquitetura;
- terá sido apresentado a três estilos complementares de arquitetura que abrangem a organização geral, decomposição modular e controle de sistema;
- compreenderá como as arquiteturas de referência são usadas para comunicar conceitos de arquitetura e para avaliar arquiteturas de sistema.

Conteúdo

11.1 Decisões de projeto de arquitetura
11.2 Organização de sistema
11.3 Estilos de decomposição modular
11.4 Modelos de controle
11.5 Arquiteturas de referência

Sistemas grandes são sempre decompostos em subsistemas que fornecem algum conjunto de serviços relacionados. O processo inicial de projeto, que consiste em identificar esses subsistemas e estabelecer um *framework* para o controle e a comunicação de subsistemas, é denominado *projeto de arquitetura*.

No modelo apresentado no Capítulo 4, o projeto de arquitetura é o primeiro estágio no processo de projeto e representa uma ligação crítica entre os processos de engenharia de projeto e de requisitos. O processo de projeto de arquitetura envolve o estabelecimento de um framework básico que identifica os principais componentes de um sistema e as comunicações entre eles.

Bass et al. (Bass, et al., 2003) explicam três vantagens de projetar e documentar explicitamente uma arquitetura de software:

1. *Comunicação de stakeholders.* A arquitetura é uma apresentação em alto nível do sistema que pode ser usada para enfocar a discussão entre os diferentes stakeholders.
2. *Análise de sistema.* Tornar a arquitetura do sistema explícita em um estágio inicial de desenvolvimento do sistema requer alguma análise. Decisões de projeto de arquitetura têm pro-

fundo efeito sobre se o sistema pode atender aos requisitos críticos, como desempenho, confiabilidade e facilidade de manutenção.

3. *Reuso em larga escala.* Um modelo de arquitetura de sistema é uma descrição compacta e administrável de como um sistema está organizado e de como os componentes operam entre si. A arquitetura de sistema é muitas vezes a mesma para sistemas com requisitos similares e, assim, pode apoiar o reuso de software em larga escala. Conforme explicarei no Capítulo 18, pode ser possível desenvolver arquiteturas de linhas de produto em que a mesma arquitetura é usada ao longo de vários sistemas relacionados.

Hofmeister et al. (Hofmeister, et al., 2000) explica como o estágio de projeto de arquitetura força os projetistas de software a considerar aspectos principais do projeto logo no início. Eles sugerem que a arquitetura de software possa servir como um plano de projeto usado para negociar requisitos de sistema e como um meio de estruturação de discussões com os clientes, desenvolvedores e gerentes. Eles também sugerem que é uma ferramenta essencial para gerenciamento de complexidade. Ela oculta detalhes e permite que os projetistas enfoquem as abstrações principais do sistema.

A arquitetura de sistema afeta o desempenho, facilidade de distribuição e de manutenção de um sistema (Bosch, 2000). O estilo e a estrutura específicos escolhidos para uma aplicação podem, portanto, depender dos requisitos não funcionais do sistema.

1. *Desempenho.* Se o desempenho for um requisito crítico, a arquitetura deve ser projetada para localizar operações críticas dentro de alguns subsistemas, com tão pouca comunicação quanto possível entre eles. Isso pode significar o uso de componentes de alta granularidade em detrimento dos de baixa granularidade para reduzir as comunicações entre os componentes.
2. *Proteção.* Se a proteção for um requisito crítico, uma estrutura em camadas para a arquitetura deve ser usada, com os itens mais críticos protegidos por camadas mais internas e com um alto nível de validação de proteção aplicado a essas camadas.
3. *Segurança.* Se a segurança for um requisito crítico, a arquitetura deve ser projetada de modo que as operações relacionadas à segurança estejam todas localizadas em um único subsistema ou em um pequeno número de subsistemas. Isso reduz os custos e os problemas de validação de segurança e torna possível fornecer esse serviço a sistemas de proteção relacionados.
4. *Disponibilidade.* Se a disponibilidade for um requisito crítico, a arquitetura deve ser projetada para incluir componentes redundantes e, assim que possível, substituir e atualizar componentes sem parar o sistema. Arquiteturas de sistema tolerantes a defeitos para sistemas de alta disponibilidade serão abordadas no Capítulo 20.
5. *Facilidade de manutenção.* Se a facilidade de manutenção for um requisito crítico, a arquitetura de sistema deve ser projetada usando componentes de baixa granularidade e autocontidos que possam ser prontamente mudados. Os criadores de dados devem ser separados dos clientes, e estruturas de dados compartilhadas devem ser evitadas.

Obviamente há conflitos potenciais entre algumas dessas arquiteturas. Por exemplo, o uso de componentes de alta granularidade aprimora o desempenho, e o uso de componentes de baixa granularidade aprimora a facilidade de manutenção. Se ambos forem requisitos de sistema importantes, alguma solução eficaz deve ser encontrada. Conforme explicado mais adiante, pode-se conseguir isso, algumas vezes, pelo uso de estilos de arquitetura diferentes para diferentes partes do sistema.

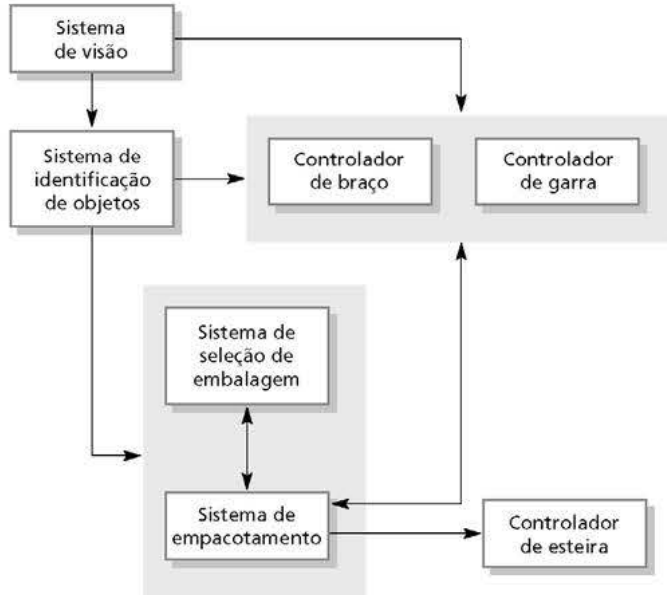
Existe uma sobreposição significativa entre os processos de engenharia de requisitos e projeto de arquitetura. Idealmente, uma especificação de sistema não deve incluir quaisquer informações de projeto. Na prática, isso é irreal exceto para sistemas muito pequenos. A decomposição de arquitetura é necessária para estruturar e organizar a especificação. Um exemplo foi apresentado no Capítulo 2, no qual a Figura 2.6 mostra a arquitetura de um sistema de controle de tráfego aéreo. Você pode usar tal modelo de arquitetura como ponto de partida para a especificação de subsistemas.

Um projeto de subsistema é uma decomposição abstrata de um sistema em componentes de alta granularidade, cada um dos quais podendo ser um sistema substancial e independente. Os diagramas de blocos são freqüentemente usados para descrever projetos de subsistemas em que cada caixa no diagrama representa um subsistema. Caixas dentro de caixas indicam que o subsistema foi ele próprio decomposto em subsistemas. As setas significam que dados e sinais de controle são passados de um subsistema para outro na direção indicada. Os diagramas de blocos apresentam um desenho de alto nível da estrutura do sistema para que pessoas de áreas diferentes envolvidas no processo de desenvolvimento do sistema possam compreender prontamente.

Por exemplo, a Figura 11.1 é um modelo abstrato de arquitetura para um sistema robotizado de empacotamento que mostra os subsistemas que precisam ser desenvolvidos. Esse sistema robotizado pode empacotar diferentes tipos de objetos. Ele usa um subsistema de visualização para apanhar os objetos de uma esteira, identificar o tipo de objeto e selecionar o tipo certo de embalagem. O sistema, depois, move os objetos da esteira para serem empacotados. Ele coloca os objetos empacotados em outra esteira. Outros exemplos de projetos de arquitetura neste nível foram mostrados no Capítulo 2 (figuras 2.5 e 2.6).

Figura 11.1

Diagrama de blocos de um sistema de controle robotizado de empacotamento.



Bass et al. (Bass, et al., 2003) dizem que diagramas simples de caixas e linhas não são representações úteis de arquitetura, pois eles não mostram a natureza dos relacionamentos entre os componentes do sistema nem as propriedades externamente visíveis dos componentes. Da perspectiva do projetista, isso está absolutamente correto. Contudo, esse tipo de modelo é eficiente para comunicação com os stakeholders de sistema e para o planejamento de projeto, pois ele não está abarrotado de detalhes. Os stakeholders, por fazerem parte do processo, podem ter uma visão abstrata do sistema. O modelo identifica subsistemas principais que serão desenvolvidos independentemente e, assim, os gerentes podem iniciar a designação de pessoas para planejar o desenvolvimento dos sistemas. Os diagramas de caixas e linhas não devem, certamente, ser a única representação de arquitetura a ser usada; no entanto, são um dos modelos úteis de arquitetura.

O problema geral da decisão de como decompor um sistema em subsistemas é difícil. Naturalmente, os requisitos de sistema são um fator principal e você deve tentar criar um projeto em que haja uma correspondência estrita entre os requisitos e os subsistemas. Isso significa que, se os requisitos mudam, essa mudança provavelmente será localizada, e não distribuída entre os vários subsistemas. No Capítulo 13, descrevo várias arquiteturas genéricas de aplicações que podem ser usadas como um ponto de partida para a identificação de subsistemas.

11.1 Decisões de projeto de arquitetura

O projeto de arquitetura é um processo criativo em que se tenta estabelecer uma organização de sistema que satisfaça os requisitos funcionais e não funcionais do sistema. Por ser um processo criativo, as atividades de processo diferem radicalmente dependendo do tipo de sistema que será desenvolvido, a origem e a experiência do arquiteto do sistema e os requisitos específicos do sistema. É, portanto, mais útil pensar no processo de projeto de arquitetura sob uma perspectiva de decisão em vez de uma perspectiva de atividade. Durante o processo de projeto de arquitetura, os arquitetos de sistema precisam tomar uma série de decisões fundamentais que afetam profundamente o sistema e o seu processo de desenvolvimento. Baseados em seu conhecimento e experiência, eles precisam responder às seguintes questões fundamentais:

1. Existe uma arquitetura genérica de aplicação que possa funcionar como um modelo para o sistema que está sendo projetado?
2. Como o sistema será distribuído ao longo de vários processadores?
3. Qual ou quais estilos de arquitetura são apropriados para o sistema?
4. Qual será a abordagem fundamental usada para estruturar o sistema?
5. Como as unidades estruturais de um sistema serão decompostas em módulos?
6. Qual estratégia será usada para controlar a operação das unidades no sistema?
7. Como o projeto de arquitetura será avaliado?
8. Como a arquitetura do sistema deve ser documentada?

Embora cada sistema de software seja único, freqüentemente sistemas de um mesmo domínio de aplicação têm arquiteturas similares que refletem os conceitos fundamentais de domínio. Essas arquiteturas de aplicação podem ser bastante genéricas, como a arquitetura de sistemas de gerenciamento de informações, ou muito mais específicas. Por exemplo, linhas de produtos de aplicações são criadas com base em um núcleo de arquitetura com variações que satisfazem os requisitos específicos do cliente. Ao projetar uma arquitetura de sistema, você precisa decidir o que seu sistema e classes mais amplas de aplicação têm em comum, e decidir quanto conhecimento dessas arquiteturas de aplicação você pode reusar. As arquiteturas genéricas de aplicações serão explicadas no Capítulo 13 e as linhas de produto de aplicações, no Capítulo 18

Para sistemas embutidos e sistemas projetados para computadores pessoais, geralmente há somente um único processador, e você precisará projetar uma arquitetura distribuída para o sistema. No entanto, sistemas muito grandes são, atualmente, sistemas distribuídos em que o software de sistema é distribuído por meio de muitos computadores diferentes. A escolha da arquitetura de distribuição é uma decisão importante que afeta o desempenho e a confiabilidade do sistema. Isso é um tópico importante e independente, abordado separadamente no Capítulo 12.

A arquitetura de um sistema de software pode ser baseada em um modelo ou estilo de arquitetura específico. Um estilo de arquitetura é um padrão de organização de sistema (Garlan e Shaw, 1993), como uma organização cliente–servidor ou uma arquitetura em camadas. O conhecimento desses estilos, suas aplicações e seus pontos fortes e fracos são importantes. Contudo, as arquiteturas da maioria dos grandes sistemas não estão de acordo com um único estilo. Partes diferentes do sistema podem ser projetadas usando estilos diferentes de arquitetura. Em alguns casos, a arquitetura geral do sistema pode ser uma arquitetura composta criada pela combinação de estilos de arquitetura diferentes.

A noção de um estilo de arquitetura de Garlan e Shaw abrange as próximas três questões de projeto. Você precisa escolher a estrutura mais apropriada, como a estrutura cliente–servidor ou em camadas, que permita atender aos requisitos do sistema. Para decompor as unidades da estrutura do sistema em módulos, você deve tomar uma decisão sobre a estratégia de decomposição de subsistemas em componentes ou módulos. As possíveis abordagens permitem que diferentes tipos de arquitetura sejam implementados. Finalmente, no processo de modelagem de controle, você deve tomar decisões sobre como a execução de subsistemas é controlada. É desenvolvido um modelo geral de relacionamentos de controle entre as partes do sistema definido. Abordo esses três tópicos nas seções 11.2 a 11.4.

A avaliação de um projeto de arquitetura é difícil, pois o verdadeiro teste de uma arquitetura recai sobre quão bem ela atende os requisitos funcionais e não funcionais depois de implantada. Contudo, em alguns casos, você pode fazer uma avaliação comparando seu projeto com modelos genéricos de arquitetura ou de referência. Abordo as arquiteturas de referência na Seção 11.5 e outras arquiteturas genéricas no Capítulo 13.

O produto do processo de projeto de arquitetura é um documento de projeto de arquitetura. Ele pode incluir várias representações gráficas do sistema junto com um texto descritivo associado. Deve ser descrito como o sistema está estruturado em subsistemas, a abordagem adotada e como cada subsistema está estruturado em módulos. Os modelos gráficos do sistema apresentam perspectivas diferentes da arquitetura. Os modelos de arquitetura que podem ser desenvolvidos podem incluir:

1. Um *modelo estático de estrutura* que mostra os subsistemas ou componentes desenvolvidos como unidades separadas.
2. Um *modelo dinâmico de processo* que mostra como o sistema está organizado em processos em tempo de execução.
3. Um *modelo de interface* que define os serviços oferecidos por cada subsistema por meio de suas interfaces públicas.
4. *Modelos de relacionamentos* que mostram os relacionamentos, tal como fluxo de dados, entre os subsistemas.
5. Um *modelo de distribuição* que mostra como os subsistemas podem ser distribuídos pelos computadores.

Alguns pesquisadores propuseram o uso de linguagens de descrição de arquiteturas (ADL – *Architectural Description Language*) para descrever arquiteturas de sistemas. Bass et al. (Bass, et al., 2003) descrevem as principais características dessas linguagens. Os elementos básicos dos ADLs são componentes e conectores, e eles incluem regras e guias para arquiteturas bem formadas. No entanto, como todas as linguagens especializadas, as ADLs podem ser somente compreendidas por especialistas em linguagens e são inacessíveis para especialistas em domínios e aplicações. Isso faz com que elas sejam difíceis de analisar com base em uma perspectiva prática. Penso que elas serão somente usadas em um pequeno número de aplicações. Modelos informais e notações como a UML (Clements, et al., 2002) permanecerão como as notações mais comumente usadas para descrição de arquiteturas.

11.2 Organização de sistema

A organização de um sistema reflete a estratégia básica usada para estruturá-lo. Você precisa tomar decisões sobre o modelo geral organizacional de um sistema com antecedência no processo de projeto de arquitetura. A organização do sistema pode refletir-se diretamente na estrutura do subsistema. Contudo, é freqüente que o modelo de subsistema inclua mais detalhes que o modelo de organização, e nem sempre há um mapeamento simples dos subsistemas para a estrutura organizacional.

Nesta seção, explico três estilos organizacionais amplamente usados. São estilos de repositório de dados compartilhados, estilo de serviços e servidores compartilhados e uma máquina abstrata ou estilo em camadas no qual o sistema é organizado como uma camada de camadas funcionais. Esses estilos podem ser usados separadamente ou juntos. Por exemplo, um sistema pode ser organizado com base em um repositório de dados compartilhados, mas pode criar camadas com base nisso para apresentar uma visão mais abstrata dos dados.

11.2.1 O modelo de repositório

Os subsistemas que constituem um sistema devem trocar informações de modo que possam trabalhar juntos eficientemente. Existem duas maneiras fundamentais pelas quais isso pode ser feito.

1. Todos os dados compartilhados são mantidos em um banco de dados que pode ser acessado por todos os subsistemas. Um modelo de sistema baseado em um banco de dados compartilhado é algumas vezes denominado *modelo de repositório*.
2. Cada subsistema mantém seu próprio banco de dados. Os dados são trocados com outros subsistemas por meio da passagem de mensagens entre eles.

A maioria dos sistemas que usam grandes quantidades de dados é organizada com base em um banco de dados ou repositório compartilhado. Esse modelo é, portanto, adequado para aplicações em que os dados são gerados por um subsistema e usados por um outro. Exemplos desse tipo de sistema incluem sistemas de comando e controle, sistemas de informações gerenciais, sistemas CAD e os conjuntos de ferramentas CASE.

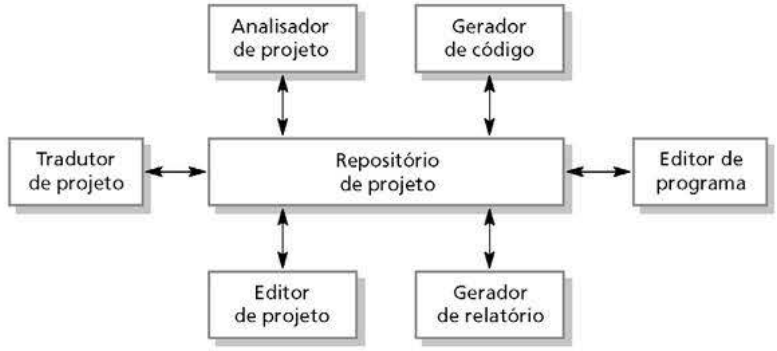
A Figura 11.2 é um exemplo de uma arquitetura de conjunto de ferramentas CASE baseado em repositório compartilhado. O primeiro repositório compartilhado para ferramentas CASE foi provavelmente desenvolvido no início da década de 1970 pela empresa inglesa ICL para apoiar o desenvolvimento de sistemas operacionais (McGuffin, et al., 1979). Esse modelo tornou-se amplamente conhecido quando Buxton (Buxton, 1980) apresentou propostas para o ambiente Stoneman para apoiar o desenvolvimento de sistemas escritos em Ada. Desde então, muitos conjuntos de ferramentas CASE foram desenvolvidos com base em um repositório compartilhado.

As vantagens e desvantagens de um repositório compartilhado são as seguintes:

1. É uma maneira eficiente de compartilhar grandes quantidades de dados. Não há necessidade de transmitir dados explicitamente de um subsistema para outro.
2. Contudo, os subsistemas devem estar de acordo com o modelo de dados do repositório. Inevitavelmente, esse é um compromisso entre as necessidades específicas de cada ferramenta. O desempenho pode ser adversamente afetado por esse compromisso. Pode ser difícil ou impossível integrar novos subsistemas se os modelos de dados não se adequarem ao esquema estabelecido.
3. Os subsistemas que produzem dados não precisam saber como esses dados são usados por outros subsistemas.
4. Contudo, a evolução pode ser difícil quando um grande volume de informações é gerado de acordo com o modelo de dados estabelecido. Traduzir isso em um novo modelo certamente será dispendioso, podendo ser difícil ou até mesmo impossível.
5. As atividades tais como back-up, proteção, controle de acesso e recuperação de erros são centralizadas. Elas são de responsabilidade do gerenciador de repositório. As ferramentas podem enfocar funções principais em vez de se preocuparem com essas questões.
6. No entanto, subsistemas diferentes podem ter requisitos diferentes para políticas de proteção, recuperação e back-up. O modelo de repositório impõe a mesma política a todos os subsistemas.

Figura 11.2

Arquitetura de um conjunto de ferramentas CASE integradas.



7. O modelo de compartilhamento é visível por meio do esquema de repositório. Novas ferramentas podem ser integradas diretamente considerando que sejam compatíveis com o modelo de dados estabelecido.
8. Contudo, pode ser difícil distribuir o repositório por uma série de máquinas. Embora seja possível distribuir um repositório logicamente centralizado, pode haver problemas com redundância e inconsistência de dados.

No modelo anterior, o repositório é passivo e o controle é de responsabilidade dos subsistemas que o usam. Uma abordagem alternativa foi derivada para sistemas de IA que usam o modelo 'blackboard', que dispara os subsistemas quando um determinado dado torna-se disponível. Isso é adequado quando a forma dos dados do repositório é bem menos estruturada. As decisões sobre qual ferramenta ativar poderá ser tomada somente quando os dados forem analisados. Esse modelo é descrito por Nii (Nii, 1986) e Bosch (Bosch, 2000), e inclui uma boa explicação de como esse estilo relaciona-se aos atributos de qualidade de sistema.

11.2.2 O modelo cliente–servidor

O modelo de arquitetura cliente–servidor é um modelo em que o sistema é organizado como um conjunto de serviços e servidores e clientes associados que acessam e usam os serviços. Os principais componentes desse modelo são:

1. Um conjunto de servidores que oferecem serviços para outros subsistemas. Exemplos disso são servidores de impressoras que oferecem serviços de impressão, servidores de arquivos que oferecem serviços de gerenciamento de arquivos e um servidor de compiladores que oferece serviços de compilação de linguagens de programação.
2. Um conjunto de clientes que solicita os serviços oferecidos pelos servidores. Esses normalmente são subsistemas independentes. Pode haver várias instâncias de um programa cliente sendo executadas simultaneamente.
3. Uma rede que permite aos clientes acessarem esses serviços. Isso não é estritamente necessário quando ambos, clientes e servidores, podem ser executados em uma única máquina. Na prática, contudo, a maioria dos sistemas cliente-servidor é implementada como sistemas distribuídos.

Os clientes talvez precisem saber os nomes dos servidores disponíveis e os serviços que eles fornecem. No entanto, os servidores não precisam saber a identidade dos clientes ou quantos clientes existem. Os clientes acessam os serviços fornecidos pelo servidor por meio de chamadas remotas de procedimento usando um protocolo request-reply, como o protocolo http usado na Web. Essencialmente, um cliente faz um pedido a um servidor e espera até receber uma resposta.

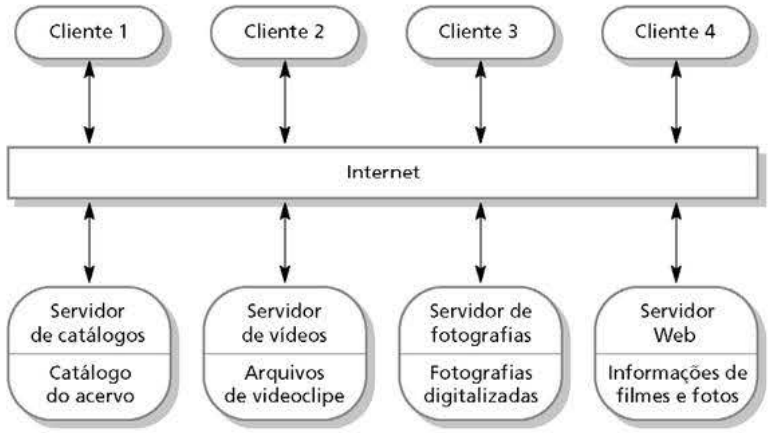
A Figura 11.3 mostra um exemplo de um sistema baseado no modelo cliente–servidor. Esse é um sistema multiusuário baseado na Web para fornecer um acervo de filmes e fotografias. Nesse sistema, vários servidores gerenciam e apresentam tipos diferentes de mídia. Os frames do vídeo precisam ser transmitidos rapidamente em sincronia, mas com uma resolução relativamente baixa. Eles podem ser comprimidos em um repositório e, assim, o servidor pode cuidar da compressão e da descompressão do vídeo em diferentes formatos. No caso de fotografias, contudo, devem ser mantidas em alta resolução; sendo assim, é apropriado mantê-las em um servidor separado.

O catálogo deve ser capaz de lidar com várias consultas e fornecer links para sistemas Web de informações com dados do filme e videoclipe, e um sistema de e-commerce que apóie a venda de filmes e videoclipes. O programa cliente é simplesmente uma interface integrada com o usuário, construída com o uso de um navegador Web para esses serviços.

A vantagem mais importante de um modelo cliente–servidor é que ele é uma arquitetura distribuída. O uso efetivo de sistemas em rede pode ser feito com muitos processadores distribuídos. É fácil adicionar um novo servidor e integrá-lo ao restante do

Figura 11.3

Arquitetura de um sistema de acervo de filmes e fotografias.



sistema ou atualizar servidores de maneira transparente sem afetar outras partes do sistema. Eu explico arquiteturas distribuídas, incluindo arquiteturas cliente–servidor e arquiteturas de objetos distribuídos, mais detalhadamente no Capítulo 12.

No entanto, mudanças em clientes e servidores existentes podem ser necessárias para obter os plenos benefícios da integração de um novo servidor. Pode não haver um modelo de dados compartilhados entre os servidores e os subsistemas podem organizar seus dados de maneiras diferentes. Isso significa que modelos de dados específicos podem ser estabelecidos em cada servidor para permitir a otimização do desempenho. Naturalmente, se uma representação de dados baseada em XML for usada, pode ser relativamente simples converter um esquema em outro. Contudo, a XML não é uma maneira eficiente de representar dados e, assim, problemas de desempenho podem surgir se ela for usada.

11.2.3 O modelo em camadas

O modelo em camadas de uma arquitetura (algumas vezes denominado modelo de máquina abstrata) organiza um sistema em camadas, cada uma das quais fornecendo um conjunto de serviços. Cada camada pode ser imaginada como uma máquina abstrata cuja linguagem de máquina é definida pelos serviços fornecidos pela camada. Essa 'linguagem' é usada para implementar o próximo nível de máquina abstrata. Por exemplo, uma maneira comum de implementar uma linguagem é definir uma 'linguagem de máquina' ideal e compilar a linguagem no código para essa máquina. Uma etapa de tradução posterior converte esse código de máquina abstrata em código de máquina real.

Um exemplo de modelo em camadas é o modelo de referência OSI de protocolos de rede (Zimmermann, 1980), explicado na Seção 11.5. Outro exemplo influente foi proposto por Buxton (Buxton, 1980), que sugeriu um modelo de três camadas para o ambiente de apoio à programação ADA (APSE – Ada Programming Support Environment). A Figura 11.4 reflete a estrutura APSE e mostra como um sistema de gerenciamento de configuração pode ser integrado usando essa abordagem de máquina abstrata.

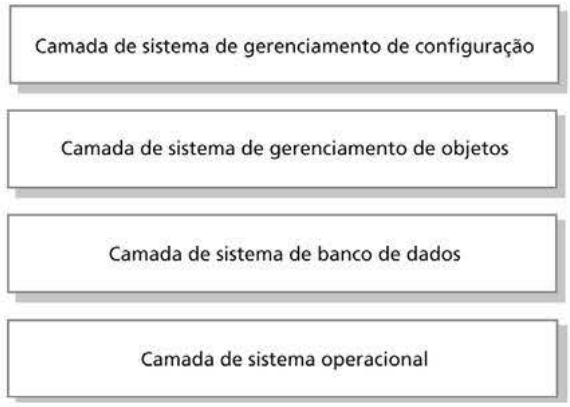
O sistema de gerenciamento de configuração gerencia versões de objetos e fornece recursos gerais de gerenciamento de configuração, como explicado no Capítulo 29. Para apoiar esses recursos de gerenciamento de configuração, é usado um sistema de gerenciamento de objetos que armazena informações e serviços de gerenciamento para itens ou objetos de configuração. Esse sistema é construído sobre um sistema de banco de dados para fornecer armazenamento e serviços básicos de dados, como gerenciamento de transações, *rollback* e recuperação, e controle de acesso. O gerenciamento de banco de dados usa recursos básicos do sistema operacional e repositório de arquivos em sua implementação. Você pode ver outros exemplos de modelos de arquitetura em camadas no Capítulo 13.

A abordagem em camadas apóia o desenvolvimento incremental de sistemas. À medida que uma camada é desenvolvida, alguns serviços fornecidos por essa camada podem ser disponibilizados para os usuários. Essa arquitetura também é modificável e portável. Desde que sua interface permaneça inalterada, uma camada poderá ser substituída por outra equivalente. Além disso, quando as interfaces de camada são alteradas, ou novos recursos são adicionados a uma camada, somente a camada adjacente é afetada. Como sistemas em camadas mantêm as dependências de máquina em camadas mais internas, isso torna mais fácil fornecer implementações de várias plataformas de um sistema de aplicação. Somente as camadas mais internas, dependentes de máquina, precisam ser reimplementadas para levar em conta os recursos de um sistema operacional ou banco de dados diferente.

Uma desvantagem da abordagem em camadas é que a estruturação de sistemas dessa maneira pode ser difícil. As camadas mais internas podem fornecer recursos básicos, como gerenciamento de arquivos, necessários em todos os níveis. Os serviços exigidos por um usuário de nível superior podem, portanto, precisar quebrar as camadas adjacentes para obter acesso aos serviços fornecidos a vários níveis abaixo dele. Isso contraria o modelo, pois a camada mais externa no sistema não depende mais de sua camada imediatamente anterior.

Figura 11.4

Modelo em camadas de um sistema de gerenciamento de versões.



O desempenho também pode ser um problema devido aos vários níveis de interpretação de comandos algumas vezes exigidos. Se houver muitas camadas, um pedido de serviço de uma camada superior precisa ser interpretado várias vezes em camadas diferentes antes de ser processado. Para evitar esses problemas, as aplicações podem ter de se comunicar diretamente com camadas mais internas em vez de usar os serviços fornecidos pela camada adjacente.

11.3 Estilos de decomposição modular

Depois que a organização geral do sistema foi escolhida, você precisa tomar uma decisão sobre a abordagem a ser usada na decomposição de subsistemas em módulos. Não há uma distinção rígida entre a organização do sistema e a decomposição modular. Os estilos explicados na Seção 11.2 podem ser aplicados neste nível. Contudo, os componentes em módulos são geralmente menores do que os subsistemas, que permitem a utilização de estilos alternativos de decomposição.

Não há uma clara distinção entre subsistemas e módulos, mas acho útil pensar neles da seguinte maneira:

1. Um subsistema é um sistema em si cuja operação não depende de serviços fornecidos por outros subsistemas. Os subsistemas são compostos de módulos e definem interfaces, as quais são usadas para comunicação com outros subsistemas.
2. Um módulo é normalmente um componente de sistema que fornece um ou mais serviços para outros módulos. Ele faz uso de serviços fornecidos por outros módulos. Não é normalmente considerado como um sistema independente. Os módulos são em geral compostos de uma série de outros componentes mais simples do sistema.

   Existem duas estratégias principais que você pode usar ao decompor um subsistema em módulos:

   - *Decomposição orientada a objetos* na qual você compõe um sistema em um conjunto de objetos que se comunicam.
   - *Pipelining* orientado a funções no qual você decompõe um sistema em módulos funcionais que aceitam dados de entrada e transforma-os em dados de saída.

Em uma abordagem orientada a objetos, os módulos são objetos com estado privado e operações definidas sobre esse estado. No modelo de *pipelining*, os módulos são transformações funcionais. Em ambos os casos, os módulos podem ser implementados como componentes ou processos seqüenciais.

Você deve evitar tomar decisões prematuras sobre a simultaneidade de um sistema. A vantagem de evitar um projeto de sistema concorrente é que programas seqüenciais são mais fáceis de projetar, implementar e testar do que sistemas paralelos. Dependências de tempo entre processos são difíceis de formalizar, controlar e verificar. É melhor decompor sistemas em módulos e, em seguida, decidir durante a implementação se eles necessitam ser executados seqüencial ou paralelamente.

11.3.1 Decomposição orientada a objetos

Um modelo de arquitetura orientada a objetos estrutura o sistema em um conjunto de objetos não firmemente acoplados com interfaces bem definidas. Os objetos chamam serviços oferecidos por outros objetos. Já apresentei modelos de objetos no Capítulo 8, e explico o projeto orientado a objetos mais detalhadamente no Capítulo 14.

A Figura 11.5 é um exemplo de modelo de arquitetura orientada a objetos de um sistema de processamento de faturas. Esse sistema pode emitir faturas aos clientes, receber pagamentos, emitir recibos para esses pagamentos e avisos de faturas não pagas. Uso a notação UML apresentada no Capítulo 8 em que as classes de objeto têm nomes e um conjunto de atributos associados. As operações são definidas na parte inferior do retângulo que representa um objeto. Setas com linhas tracejadas indicam que um objeto usa os atributos e os serviços fornecidos por um outro objeto.

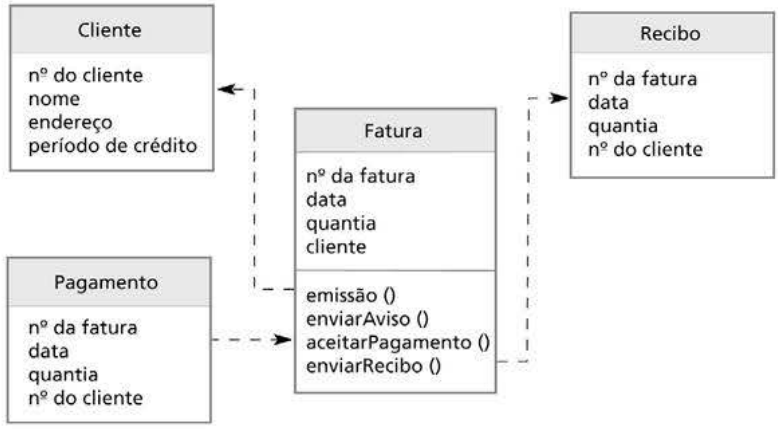
Uma decomposição orientada a objetos está relacionada a classes de objetos, seus atributos e suas operações. Quando implementados, os objetos são criados a partir dessas classes e algum modelo de controle é usado para coordenar as operações de objetos. Nesse exemplo em particular, a classe Fatura tem várias operações associadas que implementam a funcionalidade do sistema. Essa classe faz uso de outras classes que representam clientes, pagamentos e recibos.

As vantagens da abordagem orientada a objetos são bem conhecidas. Devido aos objetos não serem firmemente acoplados, a implementação de objetos pode ser modificada sem afetar outros objetos. Os objetos são muitas vezes representações de entidades do mundo real e, assim, a estrutura do sistema é rapidamente entendida. Como essas entidades do mundo real são usadas em sistemas diferentes, os objetos podem ser reusados. Linguagens de programação orientadas a objetos foram desenvolvidas de modo a permitirem implementação direta de componentes de arquitetura.

Contudo, a abordagem orientada a objetos tem desvantagens. Para usar serviços, os objetos devem fazer referência explícita ao nome e à interface de outros objetos. Se uma mudança de interface for necessária para satisfazer propostas de

Figura 11.5

Modelo de objeto de um sistema de processamento de faturas.



mudanças de sistema, o efeito dessa mudança sobre todos os usuários do objeto alterado deve ser avaliado. Enquanto os objetos podem ser mapeados claramente para entidades do mundo real em pequena escala, entidades mais complexas são algumas vezes difíceis de representar como objetos.

11.3.2 Pipelining orientado a funções

Em um pipelining orientado a funções ou modelo de fluxo de dados, as transformações funcionais processam suas entradas e produzem saídas. Os dados fluem de uma para outra função e são transformados ao moverem-se seqüencialmente. Cada etapa do processamento é implementada como uma transformação. Os dados de entrada fluem através dessas transformações até serem convertidos em saída. As transformações podem ser executadas seqüencial ou paralelamente. Os dados podem ser processados por cada transformação item por item ou em um único lote.

Quando as transformações são representadas como processos separados, esse modelo é algumas vezes chamado estilo de duto (*pipe*) e filtro, segundo a terminologia usada no sistema Unix. O sistema Unix fornece dutos que agem como condutores de dados e um conjunto de comandos que são transformações funcionais. Os sistemas que atendem a esse modelo podem ser implementados pela combinação de comandos Unix que usam dutos e os recursos do shell do Unix. O termo *filtro* é usado porque uma transformação 'filtra' os dados que pode processar a partir do caminho de dados de entrada.

Variações desse modelo de pipelining são usadas desde que os computadores foram utilizados para processamento automático de dados pela primeira vez. Quando as transformações são seqüenciais com dados processados em lotes, esse modelo de arquitetura é um modelo seqüencial em lote. Conforme explico no Capítulo 13, esse modelo é uma arquitetura comum para sistemas de processamento de dados, como sistemas de cobrança. Sistemas de processamento de dados geram normalmente muitos relatórios de saída derivados de processamentos simples sobre uma grande quantidade de registros de entrada.

Um exemplo desse tipo de arquitetura de sistema é mostrado na Figura 11.6. Uma organização emitiu faturas para os clientes. Uma vez por semana, os pagamentos feitos são compatíveis com as faturas. Para as faturas pagas, é emitido um recibo. Para as que não foram pagas dentro de um prazo permitido de pagamento, é emitido um aviso.

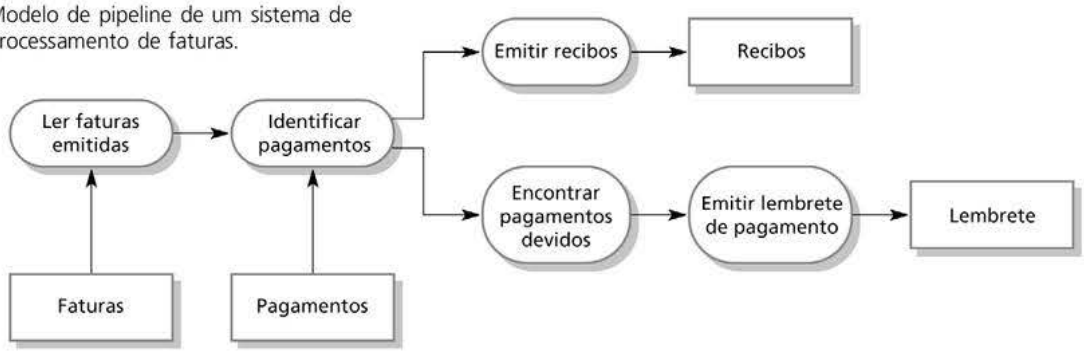
Esse é um modelo de somente uma parte do sistema de processamento de faturas; transformações alternativas seriam usadas para a emissão de faturas. Observe a diferença entre esse modelo e seu equivalente orientado a objetos explicado na seção anterior. O modelo de objeto é mais abstrato, pois não inclui informações sobre a seqüência de operações.

As vantagens dessa arquitetura são:

1. Ela apóia o reuso de transformações.
2. Ela é intuitiva, no sentido de que muitas pessoas pensam em seu trabalho em termos de processamento de entrada e saída.
3. A evolução do sistema pela adição de novas transformações é geralmente direta.
4. Ela é simples de ser implementada tanto quanto um sistema concorrente ou seqüencial.

O principal problema com esse estilo é que ele necessita ser um formato comum para transferência de dados que possa ser reconhecido por todas as transformações. Cada transformação deve estar de acordo com suas transformações de comunicação sobre o formato de dados que serão processados ou, então, deve ser imposto um formato padrão para todos os dados comunicados. Esta última é uma abordagem viável somente quando as transformações são independentes e reusáveis. No Unix, o formato-padrão é simplesmente uma seqüência de caracteres. Cada transformação deve analisar sua entrada e ajustar sua saída de acordo com a forma estabelecida. Isso aumenta o *overhead* do sistema e pode significar a impossibilidade de integrar transformações que usam formatos de dados incompatíveis.

Figura 11.6 Modelo de pipeline de um sistema de processamento de faturas.



Sistemas interativos são difíceis de escrever usando o modelo de pipelining por causa da necessidade de uma seqüência de dados ser processada. Enquanto a entrada e saída de textos simples podem ser modeladas dessa maneira, interfaces gráficas com o usuário têm formatos e controle de entrada/saída mais complexos, baseados em eventos como cliques com o mouse ou seleções de menu. É difícil traduzir isso de uma forma compatível com o modelo de pipelining.

11.4 Modelos de controle

Os modelos de estruturação estão relacionados a como um sistema é decomposto em subsistemas. Para funcionar como um sistema, os subsistemas devem ser controlados de maneira que seus serviços sejam entregues no lugar certo e no tempo certo. Modelos estruturais não incluem (nem devem incluir) informações de controle. Em vez disso, o arquiteto deve organizar os subsistemas de acordo com algum modelo de controle que suplementa o modelo de estrutura usado. Modelos de controle no nível da arquitetura lidam com o fluxo de controle entre subsistemas.

Existem dois estilos genéricos de controle usados em sistemas de software:

1. *Controle centralizado.* Um subsistema tem responsabilidade geral pelo controle e inicia e pára outros sistemas. Ele também pode passar o controle a um outro subsistema, mas esperará que essa responsabilidade de controle seja devolvida a ele.
2. *Controle baseado em eventos.* Em vez de as informações de controle serem incorporadas a um subsistema, cada subsistema pode responder a eventos gerados externamente. Esses eventos podem vir de outros subsistemas ou do ambiente do sistema.

Modelos de controle são usados em conjunto com estilos de estrutura. Todos os estilos de estrutura que expliquei podem ser implementados por meio de controle centralizado ou baseado em eventos.

11.4.1 Controle centralizado

Em um modelo de controle centralizado, um subsistema é designado como controlador de sistema e tem a responsabilidade pelo gerenciamento da execução de outros subsistemas. Modelos de controle centralizado dividem-se em duas classes, dependendo de se os subsistemas controlados forem executados seqüencial ou paralelamente.

1. *O modelo chamada–retorno.* Este é o conhecido modelo *top-down* de sub-rotina em que o controle começa no topo da hierarquia de sub-rotina e, através de chamadas de sub-rotinas, passa para os níveis mais baixos na árvore. O modelo de sub-rotina é aplicável somente a sistemas seqüenciais.
2. *O modelo gerenciador.* Este modelo é aplicável a sistemas concorrentes. Um sistema concorrente é projetado como um gerenciador de sistema e controla o início, a parada e a coordenação de outros processos do sistema. Um processo é um subsistema ou um módulo que pode ser executado paralelamente com outros processos. Uma variação desse modelo pode também ser aplicada a sistemas seqüenciais nos quais uma rotina de gerenciamento chama subsistemas específicos dependendo dos valores de algumas variáveis de estado. Isso é geralmente implementado como uma instrução 'Case'.

O modelo chamada–retorno está ilustrado na Figura 11.7. O programa principal pode chamar as Rotinas 1, 2 e 3; a Rotina 1 pode chamar as Rotinas 1.1 ou 1.2; a Rotina 3 pode chamar as Rotinas 3.1 ou 3.2 e assim por diante. Esse é um modelo de dinâmica de programa. Ele não é um modelo de estrutura; não há necessidade de a Rotina 1.1, por exemplo, ser parte da Rotina 1.

Esse modelo familiar está incorporado a linguagens de programação como C, Ada e Pascal. O controle passa de uma rotina de alto nível na hierarquia para uma rotina de baixo nível. Ele retorna, depois, para o ponto onde a rotina foi chamada. A sub-rotina sendo executada atualmente tem a responsabilidade pelo controle e pode chamar outras rotinas ou retornar o controle para seu antecessor. É um estilo inadequado de programação retornar para algum outro ponto no programa.

O modelo chamada–retorno pode ser usado no nível do módulo para controlar funções e objetos. As sub-rotinas em uma linguagem de programação chamadas por outras sub-rotinas são naturalmente funcionais. Contudo, em muitos sistemas orientados a objetos, as operações que envolvem objetos (métodos) são implementadas como procedimentos ou funções. Por exemplo, quando um objeto Java requisita um serviço para um outro objeto, ele o faz chamando um método associado.

A natureza rígida e restritiva desse modelo é, ao mesmo tempo, um ponto forte e fraco. É um ponto forte porque é relativamente simples analisar fluxos de controle e testar como o sistema responderá a entradas específicas. É um ponto fraco porque, em operação normal, é difícil lidar com as exceções.

A Figura 11.8 é uma ilustração de um modelo de controle centralizado de gerenciamento para um sistema concorrente. Esse modelo é, muitas vezes, usado em sistemas de tempo real simples sem muitas restrições rígidas de tempo. O controlador central gerencia a execução de um conjunto de processos associados a sensores e atuadores. O sistema de monitoração de edifícios explicado no Capítulo 15 usa esse modelo de controle.

O processo controlador do sistema decide quando os processos devem ser iniciados ou interrompidos dependendo das variáveis de estado do sistema. Ele verifica se outros processos produziram informações a serem processadas ou passadas a eles para processamento. O controlador geralmente está em *loop* contínuo, varrendo os sensores e outros processos em busca de eventos ou mudanças de estado. Por esse motivo, este modelo é algumas vezes chamado modelo *evento-loop*.

11.4.2 Sistemas orientados a eventos

Em modelos de controle centralizados, as decisões de controle são geralmente determinadas pelos valores de algumas variáveis de estado do sistema. Por outro lado, os modelos orientados a eventos são regidos pelos eventos gerados externamente. O termo *evento* neste contexto não significa somente um sinal binário. Pode ser um sinal que pode assumir uma gama de valores ou uma entrada de comando baseados em um menu. A distinção entre um evento e uma entrada simples é que a ocorrência do evento está fora do controle do processo que manipula esse evento.

Existem muitos tipos de sistemas orientados a eventos. Estes incluem editores em que os eventos de interface com o usuário significam comandos de edição, sistema de produção baseados em regras como os usados em IA (Inteligência

Figura 11.7

Modelo de controle chamada–retorno.

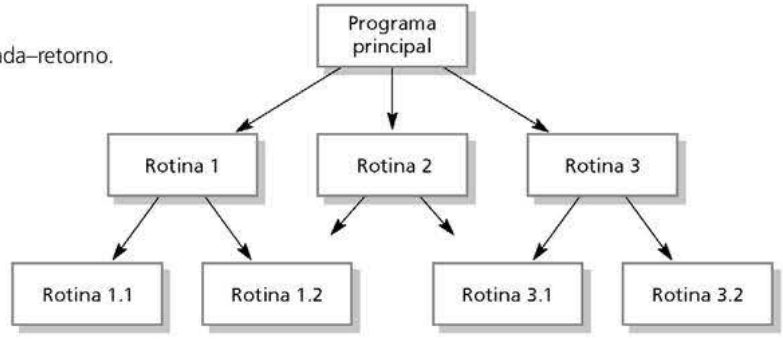
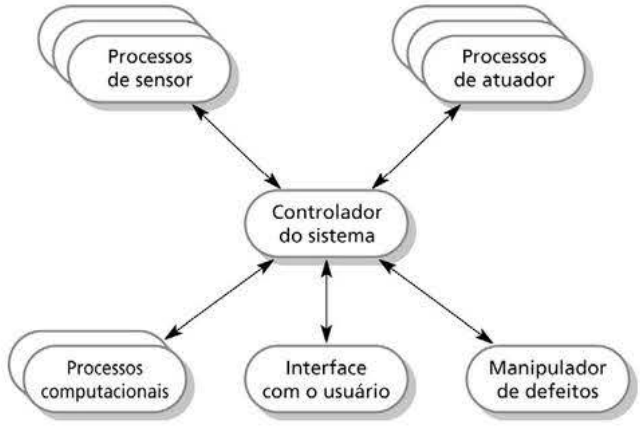


Figura 11.8

Modelo de controle centralizado para um sistema de tempo real.



Artificial), nos quais uma condição ao tornar-se verdadeira causa uma ação a ser disparada, e objetos ativos nos quais a mudança de um valor de um atributo de objeto dispara algumas ações. Garlan et al. (Garlan, et al., 1992) explicam esses diferentes tipos de sistema.

Nesta seção, abordo dois modelos de controle orientados a eventos:

1. *Modelos de broadcast*. Nesses modelos, um evento é transmitido a todos os subsistemas. Qualquer subsistema programado para manipular esse evento pode responder a ele.
2. *Modelos orientados a interrupções*. Estes modelos são usados exclusivamente em sistemas de tempo real nos quais interrupções externas são detectadas por um tratador de interrupções. Estas são, então, passadas para algum outro componente para processamento.

Modelos de broadcast são eficazes na integração de sistemas distribuídos em diferentes computadores na rede. Modelos orientados a interrupções são usados em sistemas de tempo real com requisitos rigorosos de tempo.

Em um modelo de broadcast (Figura 11.9), os subsistemas registram um interesse em eventos específicos. Quando esses eventos ocorrem, o controle é transferido para o subsistema que pode tratar o evento. A distinção entre este modelo e o modelo centralizado, mostrado na Figura 11.8, é que a política de controle não é incorporada ao tratador de eventos e mensagens. Os subsistemas decidem de quais eventos necessitam e o tratador de eventos e mensagens assegura que esses eventos sejam enviados a eles.

Todos os eventos podem ser transmitidos a todos os subsistemas, mas isso impõe um grande *overhead* de processamento. Mais freqüentemente, o tratador de eventos e mensagens mantém um registro dos subsistemas e dos eventos interessados neles. Os subsistemas geram eventos que indicam, talvez, que algum dado está disponível para processamento. O tratador de eventos detecta os eventos, consulta o registrador de eventos e passa o evento aos subsistemas que declararam interesse. Em sistemas mais simples, tais como os sistemas baseados em PC orientados a eventos de interface com o usuário, existem subsistemas explícitos do tipo evento-ouvinte que escutam os eventos do mouse, do teclado etc. e traduzem esses eventos em comandos mais específicos.

O tratador de eventos também apóia geralmente a comunicação ponto a ponto. Um subsistema pode enviar explicitamente uma mensagem a outro subsistema. Tem havido uma série de variações desse modelo, tais como o ambiente Field (Reiss, 1990) e o Softbench da Hewlett-Packard (Fromme e Walker, 1993). Ambos os modelos foram usados para controlar interações de ferramentas nos ambientes de engenharia de software. Os requisitores de objetos (ORB – Object Request Brokers), explicados no Capítulo 12, também apóiam esse modelo de controle para comunicações de objetos distribuídos.

A vantagem dessa abordagem de broadcast é que a evolução é relativamente simples. Um novo subsistema para tratar classes específicas de eventos pode ser integrado por meio do registro de seus eventos no tratador de eventos. Qualquer subsistema pode ativar qualquer outro subsistema sem saber seu nome ou sua localização. Os subsistemas podem ser implementados em máquinas distribuídas. Essa distribuição é transparente a outros subsistemas.

A desvantagem desse modelo é que os subsistemas não sabem se ou quando os eventos serão manipulados. Quando um subsistema gera um evento, ele não sabe quais outros subsistemas registraram interesse nesse evento. É bem possível que subsistemas diferentes se registrem para os mesmos eventos. Isso poderá causar conflitos quando os resultados de manipulação de eventos forem disponibilizados.

Os sistemas de tempo real que requerem que eventos gerados externamente sejam manipulados, muito provavelmente devem ser orientados a eventos. Por exemplo, se um sistema de tempo real for usado para controlar os sistemas de segurança de um carro, ele deve detectar uma possível colisão e, talvez, inflar um airbag antes que a cabeça do motorista projete-se contra o volante. Para fornecer essa rápida resposta a eventos, você precisa usar o controle orientado a interrupções.

Um modelo de controle orientado a interrupções é ilustrado na Figura 11.10. Há uma série de tipos de interrupções conhecidas com um tratador definido para cada tipo. Cada tipo de interrupção está associado ao local da memória em que o endereço de seu tratador está armazenado. Quando uma interrupção de determinado tipo é recebida, uma chave de hardware transfere o controle imediatamente para seu tratador. Esse tratador de interrupções pode, então, iniciar ou parar outros processos em resposta ao evento sinalizado pela interrupção.

Figura 11.9

Modelo de controle baseado em broadcast seletivo.

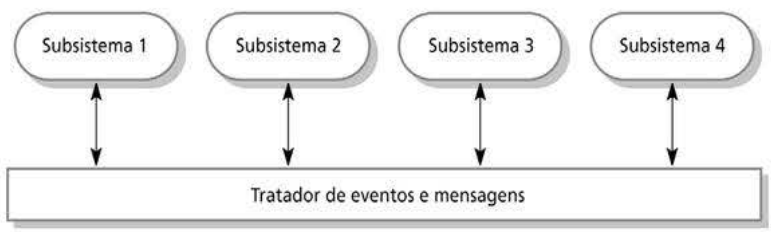
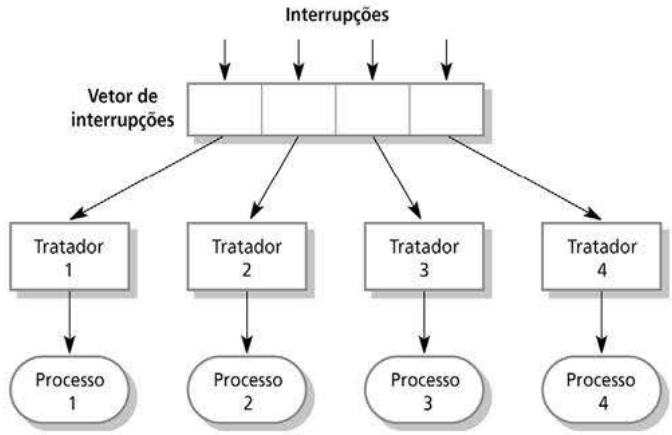


Figura 11.10

Modelo de controle orientado a interrupções.



Esse modelo é, muitas vezes, usado em sistemas de tempo real em que uma resposta imediata a algum evento é necessária. Pode ser combinado com o modelo centralizado de gerenciamento. O gerenciador central cuida da operação normal do sistema com controle baseado em interrupções para emergências.

A vantagem dessa abordagem é que ela permite respostas muito rápidas aos eventos a serem implementados. Sua desvantagem é a complexidade da programação e a dificuldade de validação. Pode ser impossível replicar padrões de tempo de interrupção durante o teste do sistema. Pode ser difícil alterar sistemas desenvolvidos usando esse modelo se o número de interrupções for limitado pelo hardware. Depois que esse limite for atingido, nenhum outro tipo de evento poderá ser manipulado. Algumas vezes, você pode obter melhores resultados com essa limitação pelo mapeamento dos vários tipos de eventos em uma única interrupção. O tratador, então, identifica qual evento ocorreu. Contudo, o mapeamento de interrupções pode não ser prático se for necessária uma resposta muito rápida para interrupções individuais.

11.5 Arquiteturas de referência

Os modelos de arquitetura anteriores são modelos gerais. Eles podem ser aplicados em muitas classes de aplicações. Assim como esses modelos gerais, modelos de arquitetura específicos a determinado domínio de aplicação podem ser usados. Embora as instâncias desses sistemas sejam diferentes em detalhes, a estrutura comum de arquitetura pode ser reusada no desenvolvimento de novos sistemas. Esses modelos de arquitetura são chamados de *arquiteturas de domínio específico*.

Existem dois tipos de modelo de arquitetura de domínio específico:

1. *Modelos genéricos* são abstrações de uma série de sistemas reais. Eles englobam as características principais desses sistemas. Por exemplo, em sistemas de tempo real, pode haver modelos genéricos de arquitetura de tipos de sistemas diferentes, como sistemas de coleta de dados ou sistemas de monitoração. Nesta seção, enfoco modelos de arquitetura de referência.
2. *Modelos de referência* são mais abstratos e descrevem uma classe maior de sistemas. Eles são um meio de informação para os projetistas sobre a estrutura geral dessa classe de sistema. Os modelos de referência são geralmente derivados de um estudo de domínio de aplicação. Eles representam uma arquitetura idealizada que inclui todas as características que esses sistemas podem incorporar.

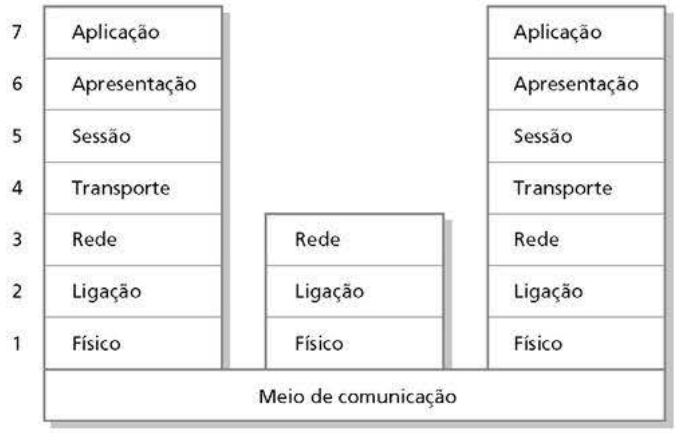
Não há, naturalmente, uma distinção rígida entre esses tipos de modelo. Modelos genéricos podem também servir como modelos de referência. Faço uma distinção entre eles neste livro porque os genéricos podem ser reusados diretamente em um projeto. Os modelos de referência são normalmente usados para comunicar conceitos de domínio e comparar ou avaliar possíveis arquiteturas.

As arquiteturas de referência não são geralmente consideradas um roteiro de implementação. Em vez disso, sua principal função é ser um meio de discussão de arquiteturas de domínio específico e de comparação de sistemas diferentes em um domínio. Um modelo de referência fornece um vocabulário para comparação. Ele atua como uma base, em relação à qual os sistemas podem ser avaliados.

O modelo OSI é um modelo de sete camadas para interconexão de sistemas abertos. O modelo é ilustrado na Figura 11.11. As funções exatas das camadas não são importantes aqui. Em essência, as camadas inferiores estão relacionadas à interconexão física; as camadas intermediárias, à transferência de dados e as camadas superiores, à transferência de informações semanticamente significativas de aplicações, como documentos padronizados.

Figura 11.11

Arquitetura de modelo de referência OSI.



Os projetistas do modelo OSI tinham um objetivo muito prático ao definir um padrão de implementação de modo que sistemas em conformidade com o padrão pudessem se comunicar uns com os outros. Cada camada deve somente depender da camada inferior a ela. À medida que a tecnologia se desenvolvesse, uma camada poderia ser novamente implementada de maneira transparente sem afetar os sistemas que usassem as outras camadas.

Na prática, contudo, os problemas de desempenho da abordagem por camadas para modelagem de arquitetura comprometeram esse objetivo. Devido a grandes diferenças entre redes, interconexões simples podem ser impossíveis. Embora as características funcionais de cada camada sejam bem definidas, as características não funcionais não estão definidas. Os desenvolvedores do sistema precisam implementar seus próprios recursos de alto nível e ignorar camadas do modelo. Como alternativa, eles precisam projetar recursos não padronizados para aprimorar o desempenho do sistema.

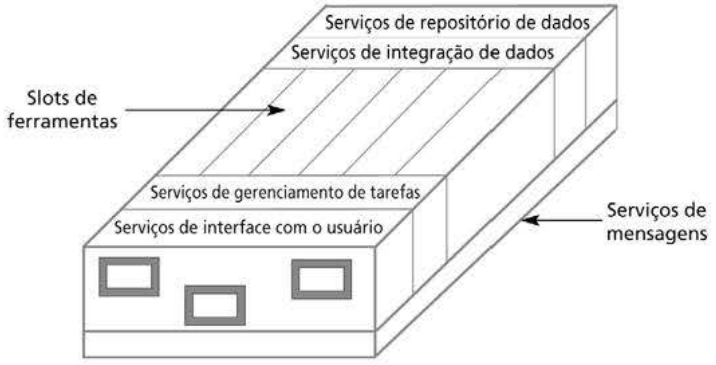
Conseqüentemente, a substituição transparente de uma camada do modelo é, na prática, quase impossível. Contudo, isso não diminui a utilidade do modelo já que fornece uma base para estruturação abstrata e a implementação sistemática de comunicações entre sistemas.

Outra proposta de modelo de referência é um modelo para ambientes CASE (ECMA, 1991; Brown, et al., 1992) que identifica cinco conjuntos de serviços que um ambiente CASE deve fornecer. Ele deve também fornecer recursos de 'plug-in' para ferramentas CASE individuais que usam esses serviços. O modelo CASE de referência é ilustrado na Figura 11.12. Os cinco níveis de serviço em um modelo CASE de referência são:

1. *Serviços de repositório de dados*. Estes serviços fornecem recursos para o armazenamento e o gerenciamento de itens de dados e seus relacionamentos.
2. *Serviços de integração de dados*. Estes serviços fornecem recursos para o gerenciamento de grupos ou para definição de relacionamentos entre eles. Estes e os serviços de repositório de dados são a base de integração de dados no ambiente.
3. *Serviços de gerenciamento de tarefas*. Estes serviços fornecem recursos para a definição e aprovação de modelos de processo. Eles apóiam a integração de processos.
4. *Serviços de mensagem*. Estes serviços fornecem recursos para as comunicações ferramenta-ferramenta, ambiente-ferramenta e ambiente-ambiente. Eles apóiam a integração de controles.
5. *Serviços de interface com o usuário*. Estes serviços fornecem recursos para o desenvolvimento de interface com o usuário. Eles apóiam a integração de apresentação.

Figura 11.12

Arquitetura de referência ECMA para ambientes CASE.



Este modelo de referência nos mostra o que pode ser incluído em qualquer ambiente CASE em particular, embora seja importante enfatizar que nem todo recurso de uma arquitetura de referência pode ser incluído nos projetos reais de arquitetura. Isso significa que não podemos formular algumas questões sobre um projeto de sistema como, por exemplo, 'como são fornecidos os serviços de repositório de dados?' e 'o sistema fornece gerenciamento de tarefas?'.

Novamente, a finalidade principal dessa arquitetura de referência é ser um meio de classificação e comparação de ferramentas e ambientes CASE integrados. Além disso, ela pode ser usada para ensinar, destacando as características principais desses ambientes, e para discuti-los de maneira genérica.

PONTOS-CHAVE

- A arquitetura de software é o framework fundamental para a estruturação do sistema. As propriedades de um sistema, como desempenho, proteção e disponibilidade são influenciadas pela arquitetura usada.
- As decisões de projeto de arquitetura incluem decisões sobre o tipo de aplicação, a distribuição do sistema, os estilos de arquitetura a serem usados e as maneiras pelas quais a arquitetura deve ser documentada e avaliada.
- Os diferentes modelos de arquitetura, como um modelo de estrutura, um modelo de controle e um modelo de decomposição podem ser desenvolvidos durante o processo de projeto de arquitetura.
- Os modelos organizacionais de um sistema incluem modelos de repositório, modelos cliente–servidor e modelos de máquina abstrata. Os modelos de repositório compartilham dados através de um repositório comum. Os modelos cliente–servidor geralmente distribuem os dados. Os modelos de máquina abstrata são estruturados em camadas, com cada camada sendo implementada por meio dos recursos da camada base.
- Os estilos de decomposição incluem decomposições orientada a objetos e orientada a funções. Os modelos de *pipelining* são funcionais e os modelos de objetos baseiam-se em entidades não firmemente acopladas que mantêm seu próprio estado e suas operações.
- Os modelos de controle incluem controle centralizado e controle baseado em eventos. Nos modelos centralizados de controle, as decisões de controle são tomadas dependendo do estado do sistema; em modelos de evento, os eventos externos controlam o sistema.
- As arquiteturas de referência podem ser usadas como um meio de discutir arquiteturas de domínio específico e avaliar e comparar projetos de arquitetura.

LEITURAS SUGERIDAS

Software architecture in practice, 2nd ed. Este livro é uma explicação prática das arquiteturas de software, não exagera na abordagem, fornece uma lógica de negócios clara e explica por que as arquiteturas são importantes. (L. Bass, et al., 2003, Addison-Wesley.)

Design and use of software architectures. Embora este livro enfoque as arquiteturas de linha de produtos, os capítulos iniciais são uma excelente introdução para tópicos gerais em projeto de arquitetura de software. (J. Bosch, 2000, Addison-Wesley.)

Software architecture: perspectives on an emerging discipline. Este foi o primeiro livro sobre arquitetura de software e traz uma boa explicação sobre estilos diferentes de arquitetura. (M. Shaw e D. Garlan, 1996, Prentice-Hall.)

EXERCÍCIOS

11.1 Explique por que pode ser necessário projetar a arquitetura de sistema antes que as especificações sejam escritas.

11.2 Explique por que podem surgir conflitos de projeto em uma arquitetura na qual os requisitos de disponibilidade e proteção são os requisitos não funcionais mais importantes.

11.3 Crie uma tabela mostrando as vantagens e as desvantagens dos modelos de estrutura explicados neste capítulo.

11.4 Sugira um modelo apropriado de estrutura para os seguintes sistemas, justificando sua resposta:

- Um sistema automático de emissão de bilhetes usado pelos passageiros de uma estação ferroviária.
- Um sistema de videoconferência controlado por computador que permita aos dados de vídeo, de áudio e de computador serem visíveis a vários participantes ao mesmo tempo.
- Um robô limpador de chão se destina a limpar espaços relativamente abertos tais como corredores. O robô deve ser capaz de identificar paredes e outros obstáculos.

11.5 Projete uma arquitetura para os sistemas anteriores baseado na sua escolha de modelo. Formule hipóteses adequadas sobre os requisitos do sistema.

11.6 Os sistemas de tempo real usam geralmente modelos de controle orientados a eventos. Sob quais circunstâncias você recomendaria o uso de um modelo de controle chamada-retorno para um sistema de tempo real?

11.7 Sugira um modelo de controle apropriado para os seguintes sistemas, justificando sua resposta:

- Um sistema de processamento em lotes que capta informações sobre horas trabalhadas e taxas de pagamentos e imprime comprovantes e informações de transferência de crédito bancário;
- Um conjunto de ferramentas de software produzidas por fabricantes diferentes, mas que devem trabalhar juntas;
- Um controlador de televisão que responde aos sinais de uma unidade de controle remoto.

11.8 Explique as vantagens e as desvantagens da facilidade de distribuição do modelo de fluxo de dados e do modelo de objetos. Suponha que são necessárias uma única máquina e versões distribuídas de uma aplicação.

11.9 Você deve comparar dois conjuntos de ferramentas CASE integradas. Explique como você pode usar um modelo de referência para CASE (Brown, et al., 1992) a fim de fazer essa comparação.

11.10 Deve haver uma profissão específica além de 'arquiteto de software' cujo papel seja trabalhar independentemente com um cliente para projetar uma arquitetura de sistema de software? Esse sistema seria então implementado por alguma empresa de software. Quais poderiam ser as dificuldades para se estabelecer tal profissão?

12

Arquiteturas de sistemas distribuídos

Objetivos

O objetivo deste capítulo é abordar modelos de arquitetura de software para sistemas distribuídos. Após ler este capítulo, você:

- compreenderá as vantagens e desvantagens de arquiteturas de sistemas distribuídos;
- compreenderá os dois principais modelos de sistemas distribuídos, denominados *sistemas cliente–servidor* e *sistemas de objetos distribuídos*;
- compreenderá o conceito de um localizador de objetos e os princípios básicos dos padrões CORBA;
- terá sido apresentado às arquiteturas ponto a ponto e orientadas a serviços como meio para implementar sistemas interorganizacionais distribuídos.

Conteúdo

12.1 Arquiteturas de multiprocessadores
12.2 Arquiteturas cliente–servidor
12.3 Arquiteturas de objetos distribuídos
12.4 Computação interorganizacional distribuída

Praticamente todos os sistemas baseados em grandes computadores atualmente são sistemas distribuídos. Um sistema distribuído é aquele em que as informações em fase de processamento são distribuídas para vários computadores, em vez de ficarem confinadas em uma única máquina. Obviamente, a engenharia de sistemas distribuídos tem muito em comum com a engenharia de software, mas existem questões específicas que devem ser levadas em conta ao se projetar esse tipo de sistema. Já apresentei algumas dessas questões na introdução à arquitetura cliente–servidor, no Capítulo 11, e abordo-as mais detalhadamente aqui.

Coulouris et al. (Coulouris, et al., 2001) explicam características importantes de sistemas distribuídos. Eles identificam as seguintes vantagens de usar uma abordagem distribuída para o desenvolvimento de sistemas:

1. *Compartilhamento de recursos*. Um sistema distribuído permite o compartilhamento de recursos de hardware e software, como discos, impressoras, arquivos e compiladores, que estão associados aos diferentes computadores em uma rede.

2. *Abertura*. Sistemas distribuídos são geralmente sistemas abertos, o que significa que são projetados com base em protocolos-padrão que permitem a equipamentos e software de fabricantes diferentes serem combinados.
3. *Concorrência*. Em um sistema distribuído, vários processos podem operar ao mesmo tempo em diferentes computadores na rede. Esses processos podem (mas não precisam) se comunicar uns com os outros durante sua operação normal.
4. *Escalabilidade*. Em princípio, pelo menos, os sistemas distribuídos são escalonáveis de modo que as capacidades de um sistema possam ser ampliadas pela adição de novos recursos para atender às novas demandas do sistema. Na prática, a ligação em rede de computadores individuais no sistema pode limitar a escalabilidade do sistema. Se muitos computadores novos forem adicionados, a capacidade da rede poderá se tornar inadequada.
5. *Tolerância a defeitos*. A disponibilidade de vários computadores e o potencial de duplicação de informações significam que os sistemas distribuídos podem ser tolerantes a algumas falhas de hardware e de software (veja o Capítulo 20). Na maioria dos sistemas distribuídos, um serviço de má qualidade pode ser fornecido quando ocorrem falhas; a perda completa do serviço tende somente a ocorrer quando há uma falha na rede.

Para sistemas organizacionais de larga escala, essas vantagens significam que os sistemas distribuídos substituíram amplamente os sistemas legados centralizados desenvolvidos nas décadas de 1980 e 1990. Contudo, comparados aos sistemas que operam com um único processador ou um cluster de processadores, os sistemas distribuídos têm uma série de desvantagens:

1. *Complexidade*. Os sistemas distribuídos são mais complexos que os sistemas centralizados. Isso faz com que seja mais difícil compreender as propriedades emergentes e testar esses sistemas. Por exemplo, em vez de o desempenho do sistema ser dependente da velocidade de execução de um processador, ele depende da largura de banda da rede e da velocidade dos processadores na rede. A movimentação de recursos de um lado para outro do sistema pode afetar radicalmente o desempenho.
2. *Proteção*. O sistema pode ser acessado a partir de vários computadores diferentes, e o tráfego na rede pode estar sujeito a interceptações. Isso faz com que seja mais difícil assegurar a integridade dos dados do sistema e que os serviços não sejam degradados por ataques do tipo negação de serviços.
3. *Gerenciamento*. Os computadores de um sistema podem ser de tipos diferentes e podem operar em versões diferentes de sistemas operacionais. Defeitos em uma máquina podem se propagar a outras máquinas com conseqüências inesperadas. Isso significa que mais esforço é necessário para gerenciar e manter o sistema em operação.
4. *Imprevisibilidade*. Como todos os usuários da Web sabem, os sistemas distribuídos são imprevisíveis em suas respostas. A resposta depende da carga total do sistema, sua organização e a carga de rede. Como tudo isso pode mudar em um curto período de tempo, o tempo de resposta para uma solicitação do usuário pode variar drasticamente de uma solicitação para outra.

O desafio é projetar o software e o hardware para fornecer os recursos de sistema distribuído desejáveis e, ao mesmo tempo, minimizar os problemas inerentes a esses sistemas. Para fazer isso, você precisa compreender as vantagens e as desvantagens de diferentes arquiteturas de sistemas distribuídos. Abordo dois tipos genéricos de arquitetura de sistemas distribuídos aqui:

1. *Arquiteturas cliente–servidor*. Nesta abordagem, podemos pensar no sistema como um conjunto de serviços fornecidos aos clientes que fazem uso desses serviços. Os servidores e clientes são tratados de maneira diferente nesses sistemas.
2. *Arquiteturas de objetos distribuídos*. Neste caso, não há distinção entre servidores e clientes, e podemos pensar no sistema como um conjunto de objetos que interagem e cuja localização é irrelevante. Não há distinção entre um provedor de serviços e um usuário desses serviços.

Tanto arquiteturas cliente–servidor como de objetos distribuídos são amplamente usadas no setor, mas a distribuição de aplicações ocorre geralmente dentro de uma única organização. A distribuição é, portanto, intra-organizacional. Explico dois outros tipos de arquitetura distribuída mais adequados à distribuição interorganizacional: arquiteturas de sistema ponto a ponto (p2p) e arquiteturas orientadas a serviços. Sistemas ponto a ponto têm sido mais usados para sistemas pessoais, mas estão começando a ser usados em aplicações de negócios. No momento em que este livro estava sendo escrito, os sistemas orientados a serviços tinham sido introduzidos, mas a abordagem orientada a serviços provavelmente se tornaria um modelo de distribuição muito significativo somente num futuro próximo.

Os componentes de um sistema distribuído podem ser implementados em diferentes linguagens de programação e operar em tipos de processadores completamente diferentes. Os modelos de dados, representação de informações e protocolos de comunicação podem ser todos diferentes. Um sistema distribuído, portanto, requer software que possa gerenciar essas partes diversas e assegurar a comunicação e troca de dados. O termo *middleware* é usado para se referir a esse software — ele se situa no meio dos diferentes componentes distribuídos do sistema.

Bernstein (Bernstein, 1996) resume os tipos de middleware disponíveis para apoiar a computação distribuída. O middleware é um software de propósito geral normalmente adquirido no mercado, em vez de escrito especialmente pelos desenvolvedores da aplicação. Exemplos de middleware são o software para gerenciamento de comunicações com banco de dados, gerenciadores de transações, conversores de dados e controladores de comunicação. Descrevo os requisitores de dados (*object request brokers*), uma classe muito importante de middleware para sistemas distribuídos, mais adiante neste capítulo.

Os sistemas distribuídos são em geral desenvolvidos com o uso de uma abordagem orientada a objetos. Esses sistemas são formados por partes independentes, não firmemente integradas, cada uma das quais podendo interagir diretamente com os usuários ou com outras partes do sistema. As partes do sistema podem ter de responder a eventos independentes. Os objetos de software refletem essas características e, assim, são abstrações naturais para os componentes de sistemas distribuídos.

12.1 Arquiteturas de multiprocessadores

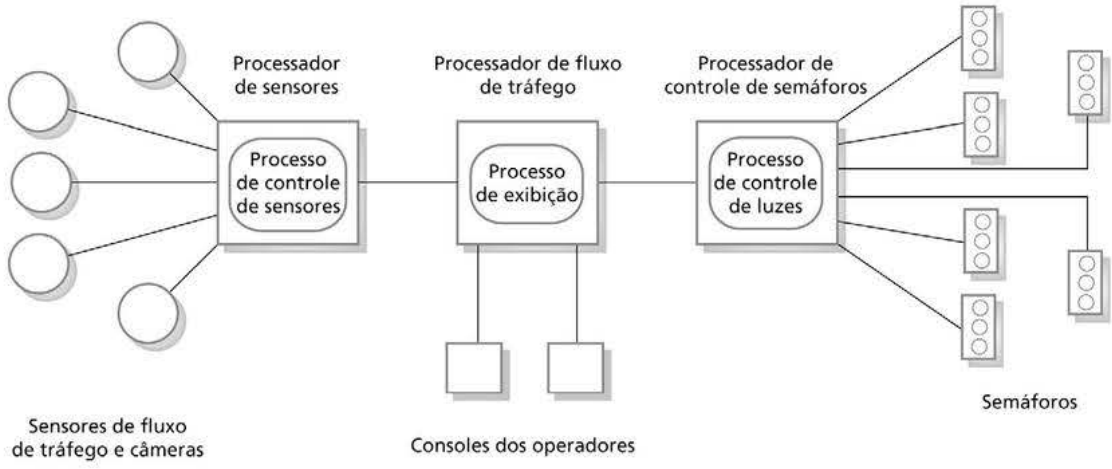
O modelo mais simples de um sistema distribuído é um multiprocessador no qual o sistema de software consiste de uma série de processos que podem (mas não precisam) ser executados em processadores separados. Esse modelo é comum em sistemas de tempo real de grande porte. Conforme explico no Capítulo 15, os sistemas coletam informações, tomam decisões usando essas informações e enviam sinais aos atuadores, que modificam o ambiente do sistema.

Logicamente, os processos dedicados à coleta de informações, à tomada de decisões e ao controle de atuadores poderiam todos ser executados em um único processador sob o controle de um escalador. O uso de vários processadores aprimora o desempenho e a capacidade de recuperação do sistema. A distribuição dos processos em processadores pode ser predeterminada (isso é comum em sistemas críticos) ou pode estar sob o controle de um *dispatcher* que decide qual processo deve ser alocado em cada processador.

Um exemplo desse tipo de sistema é mostrado na Figura 12.1. É um modelo simplificado de um sistema de controle de tráfego. Um conjunto de sensores distribuídos coleta informações do fluxo de tráfego e as processa localmente antes de enviá-las a uma sala de controle. Os operadores tomam decisões usando essas informações e dão instruções para um processo separado de controle de semáforos. Neste exemplo, existem processos lógicos separados para o gerenciamento de sensores, sala de controle e semáforos. Esses processos lógicos podem ser processos individuais ou um grupo de processos. Neste exemplo, eles são executados em processadores separados.

Os sistemas de software compostos de vários processos não são necessariamente sistemas distribuídos. Se mais de um processador estiver disponível, a distribuição poderá ser implementada, mas os projetistas do sistema não precisam sempre considerar questões de distribuição durante o processo de projeto. A abordagem de projeto para esse tipo de sistema é essencialmente a de sistemas de tempo real, conforme explicado no Capítulo 15.

Figura 12.1 Sistema multiprocessador de controle de tráfego.



12.2 Arquiteturas cliente–servidor

Já apresentei o conceito de arquiteturas cliente–servidor no Capítulo 11. Em uma arquitetura cliente–servidor, uma aplicação é modelada como um conjunto de serviços fornecidos pelos servidores e um conjunto de clientes que usam esses serviços (Orfali e Harkey, 1998). Os clientes precisam estar informados sobre os servidores disponíveis, mas

geralmente não sabem da existência de outros clientes. Os clientes e servidores são processos separados, como mostrado na Figura 12.2, que é um modelo lógico de uma arquitetura cliente–servidor distribuída.

Vários processos de servidores podem ser executados em um único processador servidor e portanto não há necessariamente um mapeamento 1:1 entre processos e processadores no sistema. A Figura 12.3 mostra a arquitetura de um sistema com seis computadores clientes e dois computadores servidores. Eles podem executar processos clientes e servidores mostrados na Figura 12.2. Quando me refiro a *clientes* e *servidores*, refiro-me a esses processos lógicos em vez de computadores físicos nos quais eles são executados.

O projeto de sistemas cliente–servidor deve refletir a estrutura lógica da aplicação que está sendo desenvolvida. Uma maneira de perceber uma aplicação está ilustrada na Figura 12.4, que mostra uma aplicação estruturada em três camadas. A camada de apresentação se encarrega da apresentação de informações para o usuário e de toda a interação com o usuário. A camada de processamento de aplicação se encarrega da implementação da lógica da aplicação e a camada de gerenciamento de dados se encarrega de todas as operações de banco de dados. Em sistemas centralizados, elas não precisam estar claramente separadas. Contudo, quando você estiver projetando um sistema distribuído, será necessário fazer uma clara distinção entre as camadas e, assim, torna-se possível distribuir cada camada para um computador diferente.

A arquitetura cliente–servidor mais simples é chamada de *arquitetura cliente–servidor de duas camadas*, na qual uma aplicação é organizada como um servidor (ou vários servidores idênticos) e um conjunto de clientes. Conforme ilustrado na Figura 12.5, as arquiteturas cliente–servidor de duas camadas podem ter duas formas:

1. *Modelo cliente-magro*. Em um modelo cliente-magro, todo o processamento da aplicação e o gerenciamento de dados são realizados no servidor. O cliente é responsável simplesmente por executar o software de apresentação.
2. *Modelo cliente-gordo*. Neste modelo, o servidor é somente responsável pelo gerenciamento de dados. O software do cliente implementa a lógica da aplicação e as interações com o usuário do sistema.

Uma arquitetura cliente-magro de duas camadas é a abordagem mais simples a ser usada quando sistemas legados centralizados, conforme explicados no Capítulo 2, evoluem para uma arquitetura cliente–servidor. A interface com o usuário desses sistemas migra para PCs, e a aplicação em si atua como um servidor e cuida de todo o processamento da aplicação e do gerenciamento de dados. Um modelo cliente-magro pode também ser implementado quando os clientes forem dispo-

Figura 12.2

Sistema cliente–servidor.

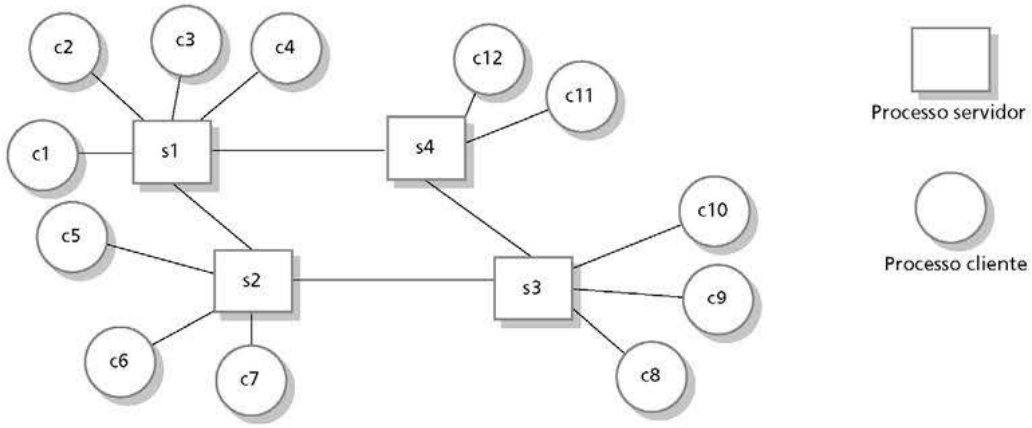


Figura 12.3

Computadores em uma rede cliente–servidor.

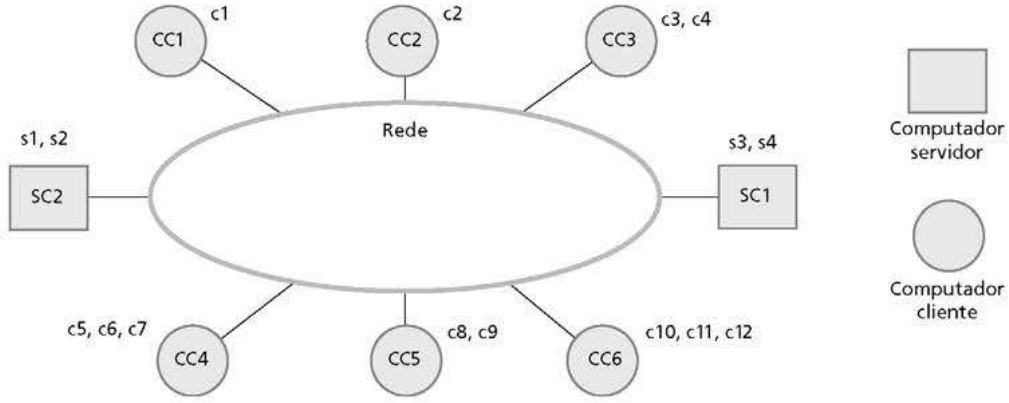


Figura 12.4

Camadas de aplicação.

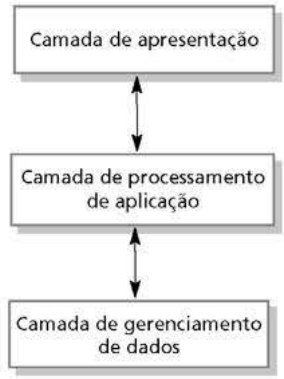
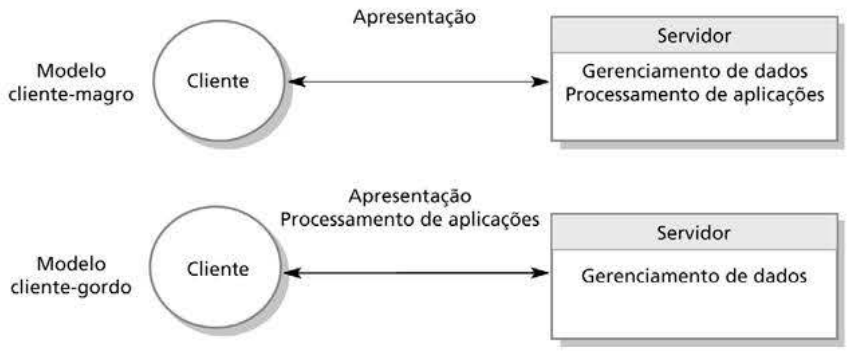


Figura 12.5

Clientes magros e gordos.



sitivos de rede simples em vez de PCs ou estações de trabalho. O dispositivo de rede opera em um navegador de Internet e a interface com o usuário é implementada por meio desse sistema.

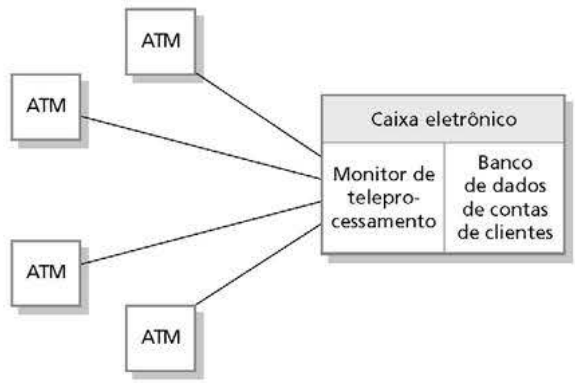
Uma desvantagem considerável do modelo cliente-magro é que ele impõe uma grande carga de processamento sobre o servidor e a rede. O servidor é responsável por todo o processamento e isso pode envolver a geração de um tráfego significativo de rede entre o cliente e o servidor. Há uma grande quantidade de capacidade de processamento disponível em dispositivos modernos de processamento e isso geralmente não é usado na abordagem cliente-magro.

O modelo cliente-gordo faz uso dessa capacidade de processamento disponível e distribui o processamento lógico de aplicação e a apresentação ao cliente. O servidor é essencialmente um servidor de transações que gerencia todas as transações de banco de dados. Um exemplo desse tipo de arquitetura são os sistemas de caixas eletrônicos de bancos, nos quais o caixa eletrônico é o cliente e o servidor é um mainframe que realiza operações sobre o banco de dados de contas dos clientes. O hardware em máquinas de atendimento realiza uma grande quantidade de processamento relacionado ao cliente associado a uma transação.

Esse sistema ATM distribuído está ilustrado na Figura 12.6. Observe que os caixas eletrônicos não se conectam diretamente ao banco de dados de clientes, mas sim a um monitor de teleprocessamento. Um monitor de teleprocessamento ou gerenciador de transações é um sistema de middleware que organiza as comunicações com os clientes remotos e coloca em série as transações de clientes para processamento pelo banco de dados. O uso de transações em série significa que o sistema pode se recuperar de defeitos sem corromper os dados de sistema.

Figura 12.6

Sistema de caixa eletrônico cliente-servidor.



Enquanto o modelo cliente-gordo distribui o processamento mais eficientemente do que um modelo cliente-magro, o gerenciamento do sistema é mais complexo. A funcionalidade da aplicação é dividida entre vários computadores. Quando o software da aplicação precisa ser alterado, isso envolve reinstalação em cada computador cliente, o que pode resultar em custo significativo se houver centenas de clientes no sistema.

O advento do código móvel (como *applets* de Java e controles Active X), que pode ser carregado de um servidor para um cliente, permitiu o desenvolvimento de sistemas cliente–servidor que estão em algum lugar entre o modelo cliente-magro e o cliente-gordo. Alguns programas de processamento de aplicação podem ser carregados para o cliente como código móvel, aliviando, portanto, a carga do servidor. A interface com o usuário é criada usando um navegador Web com recursos *built-in* para executar o código carregado.

O problema com a abordagem cliente–servidor de duas camadas é que as três camadas lógicas — apresentação, processamento de aplicações e gerenciamento de dados — devem ser mapeadas sobre dois sistemas de computador — o cliente e o servidor. Pode haver problemas de escalabilidade e desempenho, se o modelo cliente-magro for escolhido, ou problemas de gerenciamento de sistema, se o modelo cliente-gordo for usado. Para evitar esses problemas, uma abordagem alternativa é usar uma arquitetura cliente–servidor de três camadas (Figura 12.7). Nessa arquitetura, a apresentação, o processamento de aplicações e o gerenciamento de dados são processos logicamente separados executados por processadores diferentes.

Um sistema de operações bancárias pela Internet (Figura 12.8) é um exemplo de arquitetura cliente–servidor de três camadas. O banco de dados de clientes (geralmente hospedado em um computador mainframe) fornece serviços de gerenciamento de dados; um servidor Web fornece os serviços de aplicação, como recursos para transferir dinheiro, gerar extratos, pagar contas etc. e o próprio usuário do computador com um navegador de Internet é o cliente. Esse sistema é escalonável, pois é relativamente fácil adicionar novos servidores Web quando o número de clientes aumenta.

O uso de uma arquitetura de três camadas nesse caso permite a otimização da transferência de informações entre o servidor Web e o servidor de banco de dados. As comunicações entre esses sistemas podem usar protocolos rápidos de comunicações de baixo nível. Um middleware eficiente que apóia consultas de banco de dados em SQA (*Structured Query Language*) é usado para cuidar da recuperação de informações do banco de dados.

Em alguns casos, é adequado estender o modelo cliente–servidor de três camadas para uma variante com várias camadas, na qual servidores adicionais são incorporados ao sistema. Os sistemas de várias camadas podem ser usados quando as aplicações necessitam acessar e usar dados de bancos de dados diferentes. Nesse caso, um servidor de integração é colocado entre o servidor de aplicações e os servidores de banco de dados. O servidor de integração coleta os dados distribuídos e apresenta-os como se fossem provenientes de um único banco de dados.

As arquiteturas cliente–servidor de três camadas e variantes com diversas camadas que distribuem o processamento de aplicações entre os vários servidores são inerentemente mais escalonáveis do que arquiteturas de duas camadas. O tráfego de rede é reduzido em comparação com arquiteturas cliente-magro de duas camadas. O processamento de aplicações é a parte mais volátil do sistema e pode ser facilmente atualizado, pois está localizado no centro. O processamento, em alguns casos, pode ser distribuído entre os servidores de lógica de aplicações e de gerenciamento de dados, o que leva, portanto, a uma resposta mais rápida para as solicitações dos clientes.

Figura 12.7

Arquitetura cliente–servidor de três camadas.

Figura 12.8

Arquitetura de distribuição de um sistema de operações bancárias pela Internet.

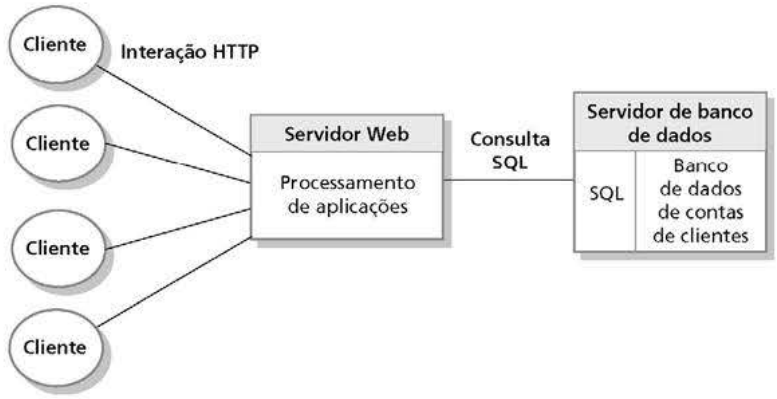


Tabela 12.1 Uso de diferentes arquiteturas cliente–servidor (C/S)

Arquitetura	Aplicações
Arquitetura C/S de duas camadas com clientes-magros	Aplicações de sistemas legados nos quais a separação de processamento de aplicações e de gerenciamento de dados é impraticável. Aplicações de computação intensiva como compiladores com pouco ou nenhum gerenciamento de dados. Aplicações de uso intensivo de dados (navegação e consultas) com pouco ou nenhum processamento de aplicações.
Arquitetura C/S de duas camadas com clientes-gordos	Aplicações nas quais o processamento é fornecido por software comercial (por exemplo, Microsoft Excel) no cliente. Aplicações nas quais o processamento de dados de computação intensiva (por exemplo, visualização de dados) é necessário. Aplicações com funcionalidade de usuário final relativamente estável usadas em um ambiente de gerenciamento de sistemas bem estabelecido.
Arquitetura C/S de três camadas ou multicamadas	Aplicações de grande escala com centenas ou milhares de clientes. Aplicações em que tanto os dados quanto as aplicações são voláteis. Aplicações em que dados de várias fontes são integrados.

Os projetistas de arquiteturas cliente–servidor devem levar em consideração uma série de fatores na escolha da arquitetura mais apropriada. Situações em que as arquiteturas cliente–servidor explicadas são provavelmente mais apropriadas são descritas na Tabela 12.1.

12.3 Arquiteturas de objetos distribuídos

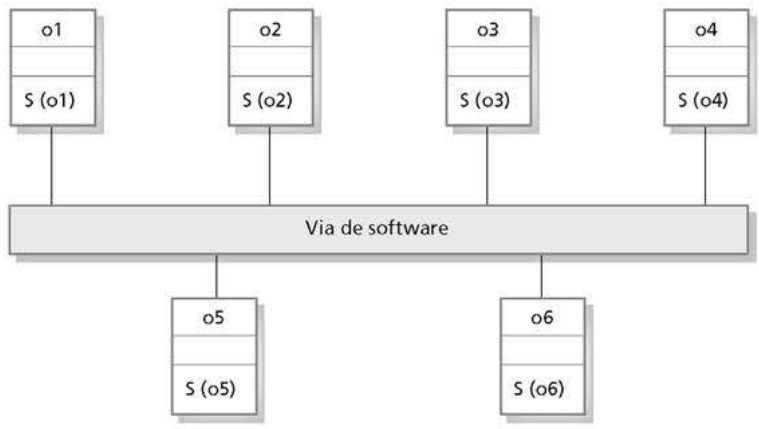
No modelo cliente–servidor de um sistema distribuído, os clientes e os servidores são diferentes. Os clientes recebem serviços dos servidores e não de outros clientes; os servidores podem atuar como clientes quanto ao recebimento de serviços de outros servidores, mas eles não solicitam serviços de clientes; os clientes devem conhecer os serviços oferecidos por servidores específicos e saber como contatar esses servidores. Esse modelo funciona bem para muitos tipos de aplicações. Contudo, ele limita a flexibilidade dos projetistas de sistema, pois eles devem decidir onde os serviços serão fornecidos. Devem também planejar a escalabilidade e, assim, fornecer algum meio para que a carga dos servidores seja distribuída quando mais clientes forem adicionados ao sistema.

Uma abordagem mais genérica para projeto de sistemas distribuídos é remover a distinção entre cliente e servidor e projetar a arquitetura do sistema como uma arquitetura de objetos distribuídos. Em uma arquitetura de objetos distribuídos (Figura 12.9), os componentes fundamentais do sistema são objetos que fornecem uma interface para um conjunto de serviços fornecidos. Outros objetos chamam esses serviços sem distinção lógica entre um cliente (receptor de um serviço) e um servidor (provedor de um serviço).

Os objetos podem ser distribuídos entre uma série de computadores na rede e se comunicam através de um middleware. Esse middleware é chamado de requisitor de objetos. Seu papel é fornecer uma interface transparente contínua entre os

Figura 12.9

Arquitetura de objetos distribuídos.



objetos. Ele fornece um conjunto de serviços que permitem que os objetos se comuniquem e sejam adicionados e removidos do sistema. Os requisitores de objetos são explicados na Seção 12.3.1.

As vantagens do modelo de objetos distribuídos são:

- Ele permite que o projetista do sistema postergue decisões sobre onde e como os serviços devem ser fornecidos. Os objetos que fornecem serviços podem ser executados em qualquer nó da rede. Portanto, a distinção entre os modelos cliente-magro e cliente-gordo torna-se irrelevante, pois não há necessidade de decidir antecipadamente onde estão localizados os objetos lógicos de aplicação.
- É uma arquitetura de sistema aberto que permite que novos recursos sejam adicionados quando necessários. Conforme explicado na próxima seção, foram desenvolvidos e implementados padrões de comunicação que permitem aos objetos escritos em linguagens diferentes de programação se comunicarem e fornecerem serviços uns aos outros.
- O sistema é flexível e escalonável. Instâncias diferentes do sistema com o mesmo serviço fornecido pelos diferentes objetos ou pelos objetos duplicados podem ser criadas para lidar com cargas diferentes do sistema. Os novos objetos podem ser adicionados quando a carga do sistema aumentar sem interromper outros objetos do sistema.
- É possível reconfigurar o sistema dinamicamente com objetos que migram através da rede conforme necessário. Isso pode ser importante onde há padrões de demanda de serviços flutuantes. Um objeto provedor de serviços pode migrar para o mesmo processador como objeto solicitador de serviço, portanto, aprimorando o desempenho do sistema.

Uma arquitetura de objetos distribuídos pode ser usada como um modelo lógico que permite estruturar e organizar o sistema. Nesse caso, você deve pensar sobre como fornecer a funcionalidade da aplicação somente em termos de serviços e combinações de serviços. Você então decide como fornecer esses serviços usando alguns objetos distribuídos. Nesse nível, os objetos projetados são, geralmente, objetos de alta granularidade (algumas vezes chamados de *objetos de negócios*) que fornecem serviços de domínio específico. Por exemplo, em uma aplicação de varejo, pode haver objetos de negócios destinados a controle de estoque, comunicações com clientes, pedidos de produtos etc. Esse modelo lógico pode assim, naturalmente, ser considerado um modelo de implementação.

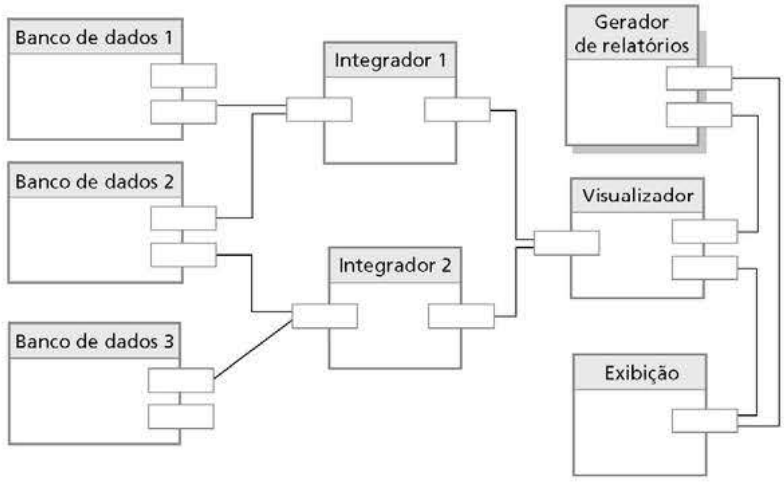
Como alternativa, você pode usar uma abordagem de objetos distribuídos para implementar sistemas cliente–servidor. Nesse caso, o modelo lógico do sistema é um modelo cliente–servidor, mas tanto clientes quanto servidores são considerados objetos distribuídos que se comunicam por uma via de software. Isso torna possível mudar o sistema facilmente, por exemplo, de um sistema de duas camadas para um com várias camadas. Nesse caso, o servidor ou o cliente não pode ser implementado como um único objeto distribuído, mas pode ser composto por objetos menores que fornecem serviços especializados.

Um exemplo desse tipo de sistema em que uma arquitetura de objetos distribuídos pode ser adequada é um sistema de *data mining* que procura relacionamentos entre os dados armazenados em uma série de banco de dados (Figura 12.10). Um exemplo de aplicação de *data mining* pode ser onde um comércio varejista tem, digamos, depósitos de alimentos e depósitos de materiais e deseja encontrar relacionamentos entre compras de clientes de alimentos e materiais. Por exemplo, pessoas que compram alimentos para bebês podem também comprar tipos diferentes de papel de parede. Com esse conhecimento, a empresa pode enfocar especificamente os clientes de alimentos para bebês com ofertas combinadas.

Nesse exemplo, cada banco de dados pode ser encapsulado como um objeto distribuído com uma interface que fornece acesso de somente leitura a seus dados. Os objetos de integração são dedicados a tipos específicos de relacionamentos e coletam informações de todos os bancos de dados para tentar deduzir os relacionamentos. Pode haver um objeto de inte-

Figura 12.10

Arquitetura de distribuição de um sistema de mineração de dados.



gração dedicado às variações sazonais de vendas de produtos e um outro dedicado aos relacionamentos entre diferentes tipos de produtos.

Os objetos de visualização interagem com os objetos de integração para produzir uma visualização ou um relatório sobre os relacionamentos descobertos. Devido aos grandes volumes de dados tratados, os objetos de visualização usarão normalmente apresentações gráficas de relacionamentos descobertos. Explico a apresentação gráfica de informações no Capítulo 16.

Uma arquitetura de objetos distribuídos em lugar de uma arquitetura cliente–servidor é adequada para esse tipo de aplicação por três razões:

1. Diferente de um caixa eletrônico bancário, por exemplo, o modelo lógico do sistema não é um dos fornecimentos de serviço em que existem serviços distintos de gerenciamento de dados.
2. Você pode adicionar bancos de dados ao sistema sem grande interrupção. Cada banco de dados é simplesmente um outro objeto distribuído. Os objetos de banco de dados podem fornecer uma interface simplificada que controla o acesso aos dados. Os bancos de dados acessados podem residir em máquinas diferentes.
3. Ela permite que novos tipos de relacionamentos sejam extraídos pela adição de novos objetos de integração. As partes do negócio interessadas em relacionamentos específicos podem ampliar o sistema pela adição de objetos de integração que operam em seus computadores sem exigir conhecimento de quaisquer outros integradores usados em qualquer lugar.

A maior desvantagem de arquiteturas de objetos distribuídos é que elas são mais complexas de projetar do que sistemas cliente–servidor. Os sistemas cliente–servidor parecem ser uma maneira bastante natural de pensar sobre sistemas. Eles refletem muitas transações humanas em que as pessoas solicitam e recebem serviços de outras pessoas que se especializam em fornecer esses serviços. É mais difícil pensar sobre fornecimento geral de serviços, e não temos ainda uma grande experiência com o projeto e o desenvolvimento de objetos de negócios de alta granularidade.

12.3.1 CORBA

Conforme indiquei na seção anterior, a implementação de uma arquitetura de objetos distribuídos requer um middleware (requisitor de objetos) para cuidar das comunicações entre os objetos distribuídos. Em princípio, os objetos do sistema podem ser implementados por meio de diferentes linguagens de programação, os objetos podem ser executados em diferentes plataformas e seus nomes não precisam ser conhecidos por todos os outros objetos do sistema. O middleware, portanto, precisa fazer um grande trabalho para assegurar as comunicações contínuas de objetos.

Para apoiar a computação de objetos, o middleware é necessário em dois níveis:

1. No nível de comunicação lógica, o middleware fornece funcionalidade que permite aos objetos de diferentes computadores trocarem informações de dados e de controle. Padrões como CORBA e COM (Pritchard, 1999) foram desenvolvidos para facilitar as comunicações lógicas de objetos em plataformas diferentes.
2. No nível de componente, o middleware fornece uma base para o desenvolvimento de componentes compatíveis. Padrões como componentes EJB, CORBA ou Active X (Szyperski, 2002) fornecem uma base para a implementação de componentes com métodos padronizados que podem ser consultados e usados por outros componentes. Os padrões de componentes são abordados no Capítulo 19.

Nesta seção, enfoco o middleware para comunicação lógica de objetos e explico como isso é apoiado pelos padrões CORBA. Estes foram definidos pelo Object Management Group (OMG), que define padrões para o desenvolvimento orientado a objetos. Os padrões OMG estão disponíveis, gratuitamente, no seu site da Web.

A visão da OMG sobre uma aplicação distribuída é mostrada na Figura 12.11, adaptada do diagrama de Siegel de Arquitetura de Gerenciamento de Objetos (Object Management Architecture) (Siegel, 1998). Ele propõe que uma aplicação distribuída deve ser formada por uma série de componentes:

1. Objetos de aplicações projetados e implementados para essa aplicação.
2. Objetos padronizados definidos pelo OMG para um domínio específico. Esses padrões de domínio de objetos abrangem as áreas de finanças/seguros, comércio eletrônico, assistência médica e várias outras.
3. Os serviços fundamentais do CORBA que fornecem serviços de computação básica distribuída, como gerenciamento de diretórios e proteção.
4. Os recursos horizontais do CORBA, como recursos de interface com o usuário, de gerenciamento de sistema etc. O termo *recursos horizontais* sugere que esses recursos são comuns a vários domínios de aplicação e são, portanto, usados em muitas aplicações diferentes.

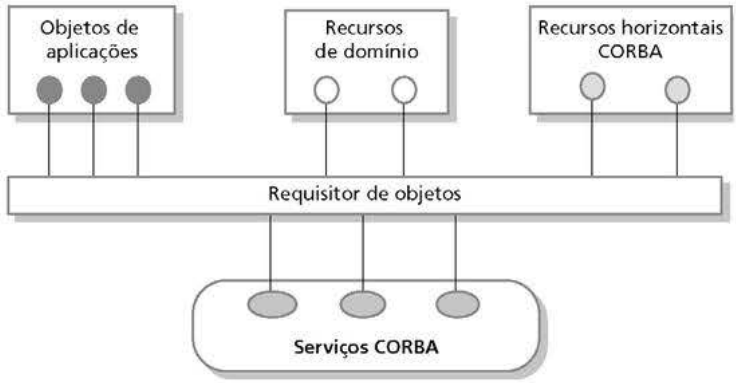


Figura 12.11

Estrutura de uma aplicação distribuída baseada em CORBA.

Os padrões CORBA abrangem todos os aspectos dessa visão. Existem quatro elementos principais desses padrões:

1. Um modelo de objeto para objetos de aplicação em que um objeto CORBA é um encapsulamento de estado com uma interface bem definida em linguagem neutra descrita em uma Linguagem de Definição de Interface (IDL — *Object Management Architecture*).
2. Um requisitor de objetos (ORB — *Object Request Broker*) que gerencia solicitações para serviços de objetos. O ORB localiza o objeto que fornece o serviço, prepara-o para uma solicitação, envia o serviço solicitado e retorna o resultado ao solicitante.
3. Um conjunto de serviços de objetos que são serviços gerais com probabilidade de serem solicitados por muitas aplicações distribuídas. Por exemplo, serviços de diretórios, serviços de transações e serviços de persistência.
4. Um conjunto de componentes comuns construídos sobre esses serviços básicos que podem ser solicitados pelas aplicações. Esses componentes podem ser verticais de domínio específico ou horizontais de próposito geral usados por muitas aplicações.

O modelo de objeto CORBA considera um objeto como se fosse um encapsulamento de atributos e serviços, como é normal em objetos. Contudo, os objetos CORBA devem ter uma definição de interface separada que define os atributos e operações públicas do objeto. As interfaces de objetos CORBA são definidas por meio de uma linguagem de definição de interface padronizada independente de linguagem. Se um objeto deseja usar serviços fornecidos por um outro objeto, ele acessa esses serviços através da interface IDL. Os objetos CORBA têm um único identificador chamado de Referência de Objeto Interoperável (IOR — *Interoperable Object Reference*). Esse IOR é usado quando um objeto solicita serviços de um outro objeto.

O requisitor de objetos conhece os objetos que estão solicitando serviços e suas interfaces. O ORB cuida da comunicação entre os objetos. Os objetos que se comunicam não precisam conhecer a localização de outros objetos nem sobre sua implementação. Quando a interface IDL isola os objetos do ORB, é possível mudar a implementação do objeto de um modo totalmente transparente. A localização do objeto pode mudar entre as chamadas, e isso é transparente para outros objetos do sistema.

Por exemplo, na Figura 12.12, dois objetos, **o1** e **o2**, se comunicam através de um ORB. O objeto chamador (**o1**) tem um *stub* de IDL associado que define a interface do objeto que fornece o serviço solicitado. O desenvolvedor do objeto **o1** inclui chamadas para esse *stub* em sua implementação de objeto quando um serviço é solicitado. A IDL é um superconjunto de C++ e, assim, é muito fácil acessar esse *stub* se você estiver programando em C++, e é relativamente fácil em C ou Java. Mapeamentos de linguagem para IDL foram também desenvolvidos para outras linguagens, como Ada e COBOL.

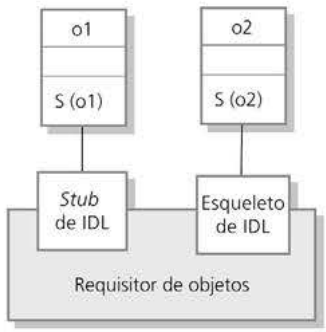


Figura 12.12

Comunicações de objetos por meio de um ORB.

O objeto que fornece o serviço tem um esqueleto de IDL associado que liga a interface à implementação dos serviços. Em termos simples, quando um serviço é chamado pela interface, o esqueleto de IDL traduz isso em uma chamada para o serviço em qualquer linguagem de implementação usada. Quando o método ou procedimento for executado, o esqueleto de IDL traduzirá os resultados em IDL e, dessa maneira, eles poderão ser acessados pelo objeto chamador. Quando um objeto fornece serviços para outros objetos e usa os serviços fornecidos em algum lugar, ele precisa de um esqueleto e *stubs* de IDL. É necessário um *stub* de IDL para cada objeto usado.

Os requisitores de objetos não são normalmente implementados como processos separados, mas são um conjunto de bibliotecas que podem ser ligadas com implementações de objetos. Portanto, em um sistema distribuído, cada computador que processa objetos distribuídos terá seu próprio requisitor de objetos. Ele cuidará de todas as chamadas locais de objetos. No entanto, uma solicitação feita para um serviço a ser fornecido pelo objeto remoto requer comunicações inter-ORB.

Essa situação está ilustrada na Figura 12.13. Nesse exemplo, quando o objeto **o1** ou **o2** solicita um serviço de **o3** ou **o4**, os ORBs associados devem se comunicar. Uma implementação CORBA apóia a comunicação ORB para ORB com o fornecimento de todos os ORBs com acesso às definições de interface de IDL e pela implementação de padrões da OMG do tipo Protocolo Inter-ORB Genérico (GIOP — Generic Inter-ORB Protocol). Esse protocolo define mensagens padronizadas que os ORBs podem trocar para implementar chamadas de objetos remotos e transferir informações. Quando combinado com protocolos de Internet de baixo nível, o GIOP permite que os ORBs se comuniquem pela Internet.

A iniciativa CORBA foi desenvolvida desde 1980 e as versões recentes de CORBA foram simplesmente dedicadas ao apoio aos objetos distribuídos. Contudo, como os padrões foram desenvolvidos, eles se tornaram mais extensos. Assim como um mecanismo para comunicações de objetos distribuídos, os padrões CORBA agora definem alguns serviços padronizados que podem ser fornecidos para apoiar as aplicações distribuídas orientadas a objetos.

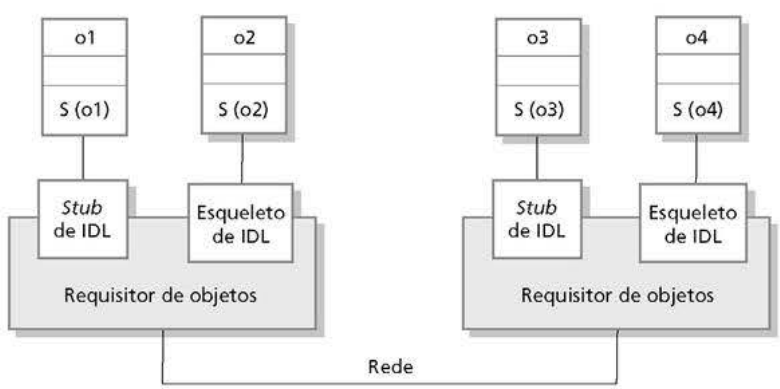
Você pode pensar nos serviços CORBA como recursos com probabilidade de serem solicitados por muitos sistemas distribuídos. Os padrões definem aproximadamente 15 serviços comuns. Alguns exemplos desses serviços genéricos são:

1. Serviços de definição de nomes e de negócio que permitem que os objetos se refiram a outros e os descubram na rede. O serviço de definição de nomes é um serviço de diretório que permite aos objetos serem denominados e descobertos por outros objetos. Isso é semelhante às páginas de uma lista telefônica. Os serviços de negociação são como páginas amarelas. Os objetos podem encontrar o que os outros objetos registraram no serviço de negociação e podem acessar a especificação desses objetos.
2. Os serviços de notificação que permitem aos objetos notificar outros objetos de que algum evento ocorreu. Os objetos podem registrar seu interesse em determinado evento no serviço e, quando esse evento ocorre, eles são automaticamente notificados. Por exemplo, digamos que o sistema inclua um *spooler* de impressora que enfileira documentos a serem impressos e uma série de objetos de impressora. O *spooler* de impressora registra que está interessado em um evento 'fim-de-impressão' de um objeto de impressora. O serviço de notificação informará quando a impressão for concluída e poderá, depois, programar o próximo documento na impressora.
3. Serviços de transação que apóiam transações atômicas e *rollback* em caso de falhas. As transações são um recurso de tolerância a defeitos que apóia a recuperação de erros durante uma operação de atualização. Se a operação de atualização de objeto falhar, então o estado do objeto poderá ser retornado ao estado anterior ao início da atualização.

Os padrões CORBA incluem definições de interface para uma grande variedade de componentes horizontais e verticais que podem ser usados pelos desenvolvedores de aplicações distribuídas. Os componentes verticais são componentes específicos de um domínio de aplicação. Os componentes horizontais são componentes de propósito geral, como componentes de interface com o usuário. O desenvolvimento de especificações para componentes CORBA horizontais e verticais é uma atividade de longo prazo e as especificações atualmente disponíveis estão publicadas no site Web da OMG.

Figura 12.13

Comunicações inter-ORBs.



12.4 Computação interorganizacional distribuída

Por motivos de proteção e interoperabilidade, a computação distribuída foi implementada inicialmente em nível organizacional. Uma organização tem uma série de servidores e distribui sua carga computacional entre eles. Devido ao fato de eles estarem todos localizados dentro da mesma organização, podem ser aplicados padrões e processos operacionais locais. Embora, para sistemas baseados na Web, os computadores clientes estejam muitas vezes fora dos limites da organização, sua funcionalidade é limitada a executar um software de interface com o usuário.

Modelos mais novos de computação distribuída, contudo, estão atualmente disponíveis e permitem computação interorganizacional distribuída em vez de intra-organizacional. Essas duas abordagens são explicadas nesta seção. A computação ponto a ponto baseia-se em computações realizadas por individuais nós de rede. Sistemas orientados a serviços contam com serviços distribuídos em vez de objetos distribuídos, e contam com padrões baseados em XML para trocas de dados.

12.4.1 Arquiteturas ponto a ponto

Sistemas ponto a ponto (p2p) são sistemas descentralizados em que as computações podem ser realizadas por qualquer nó da rede e, em princípio pelo menos, nenhuma distinção é feita entre clientes e servidores. Nas aplicações ponto a ponto, o sistema global é projetado para beneficiar-se da capacidade computacional e armazenamento disponíveis em uma rede de computadores potencialmente grande. Os padrões e protocolos que possibilitam as comunicações através dos nós estão embutidos na aplicação, e cada nó deve realizar uma cópia dessa aplicação.

No momento em que este livro estava sendo escrito, as tecnologias ponto a ponto estavam sendo mais usadas para sistemas pessoais (Oram, 2001). Por exemplo, sistemas de compartilhamento de arquivos baseados em protocolos Gnutella e Kazaa são usados para compartilhar arquivos em PCs de usuários, e sistemas de mensagens instantâneas, como ICQ e Jabber, permitem comunicação direta entre os usuários sem um servidor intermediário. SETI@home é um projeto em andamento para processar dados de radiotelescópios em PCs caseiros para buscar indicações de vida extraterrestre, e Freenet é um banco de dados descentralizado projetado para tornar mais fácil a publicação anônima de informações e tornar mais difícil para as autoridades suprimirem essas informações.

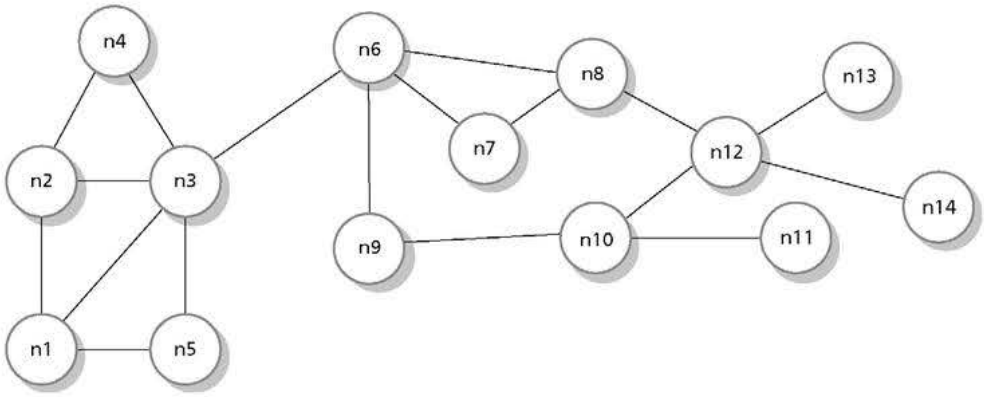
Contudo, existem indicações de que essa tecnologia está cada vez mais sendo usada pelas empresas para aproveitar a potência de suas redes de PC (McDougall, 2000). A Intel e a Boeing implementaram sistemas p2p para aplicações que requerem computação intensa. Para aplicações cooperativas que suportam trabalhos distribuídos, essa parece ser a tecnologia mais eficiente.

Você pode olhar a arquitetura das aplicações p2p de duas perspectivas. A arquitetura lógica de rede é a arquitetura de distribuição do sistema, uma vez que a arquitetura de aplicação é a organização genérica dos componentes em cada tipo de aplicação. Neste capítulo, enfoco dois tipos principais de arquitetura lógica de rede que podem ser usados — arquiteturas descentralizadas e semicentralizadas.

Em princípio, em sistemas ponto, a ponto todo nó da rede pode estar ciente a respeito de qualquer outro nó, pode fazer conexões com ele e pode trocar dados com ele. Na prática, naturalmente, isso é impossível e, assim, os nós são organizados em 'localidades' com alguns nós atuando como pontes para outras localidades de nós. A Figura 12.14 mostra essa arquitetura p2p descentralizada.

Em uma arquitetura descentralizada, os nós da rede não são simplesmente elementos funcionais, mas também chaves de comunicação que podem guiar os sinais de dados e de controle de um nó para o outro. Por exemplo, suponha que a Figura 12.14 representa um sistema descentralizado de gerenciamento de documentos. Esse sistema é usado por um consórcio de pesquisadores que compartilham documentos e cada membro do consórcio mantém seu próprio repositório de documentos.

Figura 12.14
Arquitetura p2p descentralizada.



No entanto, quando um documento é recuperado, o nó que está recuperando esse documento também o torna disponível para outros nós. Alguém que necessite de um documento emite um comando de pesquisa que é enviado aos nós naquela 'localidade'. Esses nós verificam se eles têm o documento e, caso tenham, retornam o documento ao solicitante. Se não o tiverem, eles guiarão o documento de volta ao solicitante original. Portanto, se **n1** emite uma busca por um documento armazenado em **n10**, essa pesquisa é guiada através dos nós **n3**, **n6**, e **n9** e **n10**.

Essa arquitetura descentralizada tem vantagens óbvias, pois é altamente redundante e, assim, tolerante a defeitos e tolerante quanto aos nós que se desconectam da rede. Contudo, existem overheads óbvios no sistema, pois a mesma busca pode ser processada por muitos nós diferentes e há um overhead significativo em comunicações replicadas entre pares. Uma alternativa ao modelo de arquitetura p2p originado de uma arquitetura p2p pura é uma arquitetura semicentralizada na qual, dentro da rede, um ou mais nós atuam como servidores para facilitar as comunicações entre os nós. A Figura 12.15 ilustra esse modelo.

Em uma arquitetura semicentralizada, o papel de um servidor é auxiliar a estabelecer contato entre os pares na rede ou coordenar os resultados de uma computação. Por exemplo, se a Figura 12.15 representa um sistema de mensagens instantâneas, então os nós da rede se comunicam com o servidor (indicado pelas linhas tracejadas) para encontrar quais outros nós estão disponíveis. Uma vez que esses nós sejam descobertos, comunicações diretas podem ser estabelecidas e a conexão com o servidor é desnecessária. Portanto, os nós **n2**, **n3**, **n5** e **n6** estão em comunicação direta.

Em um sistema computacional p2p no qual uma computação intensiva de processador é distribuída entre um grande número de nós, é normal que alguns — nós cujo papel é distribuir trabalho para outros nós e comparar e verificar os resultados da computação — destaquem-se.

Embora haja overheads em sistemas ponto a ponto, essa é uma abordagem muito mais eficiente para computação interorganizacional que a abordagem baseada em serviços, a qual explico na próxima seção. Existem ainda problemas com o uso de abordagens p2p para computação interorganizacional, pois problemas como proteção e confiança ainda não foram resolvidos. Isso significa que sistemas p2p são mais adequados a sistemas de informações não críticos ou nos quais já existam relacionamentos de trabalho entre as organizações.

12.4.2 Arquitetura de sistema orientada a serviços

O desenvolvimento da Web promoveu o acesso de computadores clientes aos servidores remotos fora de suas próprias organizações. Se essas organizações convertiam suas informações em HTML, estas poderiam ser acessadas por esses computadores. Contudo, o acesso era somente por meio de navegador Web, e o acesso direto aos repositórios de informações por outros programas não era prático. Isso impossibilitou as conexões convenientes entre os servidores nas quais, por exemplo, um programa consulta uma série de catálogos.

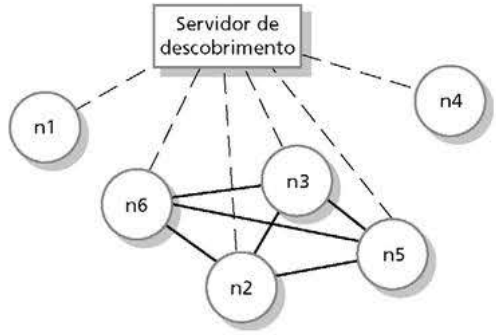
Para contornar esse problema, a noção de um Web service foi proposta. Usando um serviço deste tipo, as organizações que desejam tornar acessíveis suas informações a outros programas podem fazer isso com a definição e publicação de uma interface de Web service. Essa interface define os dados disponíveis e como eles podem ser acessados. Genericamente, um Web service é uma representação padronizada de alguns recursos computacionais e de informações que podem ser usadas por outros programas. Portanto, você pode definir um serviço de declaração de impostos no qual os usuários podem preencher seus formulários de impostos e estes serem verificados automaticamente e enviados às autoridades fiscais.

Um Web service é uma instância de uma noção mais geral de um serviço, que é definido por Lovelock et al. (Lovelock, et al., 1996) como:

> *uma ação ou desempenho oferecido de um grupo para um outro. Embora o processo possa estar ligado a um produto físico, o desempenho é essencialmente intangível e não resulta normalmente em propriedade de algum dos fatores de produção.*

Figura 12.15

Arquitetura p2p semicentralizada.



A essência de um serviço, portanto, é que o fornecimento dos serviços é independente da aplicação que usa o serviço (Turner, et al., 2003). Os provedores de serviços podem desenvolver serviços especializados e oferecê-los a uma gama de usuários de serviços de organizações diferentes. As aplicações podem ser construídas pela ligação de serviços de vários provedores que usam uma linguagem-padrão de programação ou uma linguagem de harmonização de serviços especializados como BPEL4WS.

Existem vários modelos de serviços, do modelo JINI (Kumaran, 2001) a Web services (Stal, 2002), e *Grid services* (Foster, et al., 2002). Conceitualmente, todos estes funcionam de acordo com o modelo mostrado na Figura 12.16, que é uma generalização do modelo conceitual de serviços Web descrito por Kreger (Kreger, 2001). Um provedor de serviços oferece um serviço pela definição de sua interface e pela implementação da funcionalidade do serviço. Um solicitante do serviço vincula esse serviço a sua aplicação. Isso significa que a aplicação do solicitante inclui códigos para chamar o serviço e processar os resultados da chamada. Para assegurar que o serviço possa ser acessado por usuários de serviços externos, o provedor de serviços faz uma entrada em um registro de serviço que inclui informações sobre o serviço e o que ele faz.

As diferenças entre esse modelo de serviço e a abordagem de objetos distribuídos para arquiteturas de sistemas distribuídos são:

- Os serviços podem ser oferecidos por qualquer provedor de serviços dentro ou fora de uma organização. Supondo que esses serviços estejam de acordo com certos padrões (explicados a seguir), as organizações podem criar aplicações por meio da integração de serviços de vários provedores. Por exemplo, uma empresa de manufatura pode se vincular diretamente aos serviços fornecidos por seus fornecedores.
- O provedor de serviços torna públicas as informações sobre o serviço de maneira que os usuários autorizados podem usá-lo. O provedor de serviços e o usuário do serviço não precisam negociar a respeito do que o serviço faz antes que possa ser incorporado em um programa de aplicação.
- As aplicações podem atrasar a vinculação dos serviços até que eles sejam implantados ou executados. Portanto, uma aplicação que usa um serviço de preço de ações, por exemplo, pode mudar dinamicamente os provedores de serviços enquanto o sistema estiver em execução.
- A criação conveniente de novos serviços é possível. Um provedor de serviços pode reconhecer novos serviços que podem ser criados pela vinculação de serviços existentes de maneiras inovadoras.
- Os usuários de serviços podem pagar pelos serviços de acordo com o uso no local de fornecimento. Portanto, em vez de comprar um componente caro que é usado raramente, o programador da aplicação pode usar um serviço externo que será pago somente quando necessário.
- As aplicações podem se tornar menores (o que é importante se elas forem incorporadas a outros dispositivos), pois podem implementar o tratamento de exceções como serviço externo.
- As aplicações podem ser reativas e adaptar sua operação de acordo com o ambiente por meio da vinculação de serviços diferentes à medida que o ambiente muda.

No momento em que este livro estava sendo escrito, essas vantagens desencadearam um grande interesse sobre os Web services como uma base para construção de aplicações distribuídas não firmemente acopladas. No entanto, há ainda uma experiência prática limitada quanto a arquiteturas orientadas a serviços e, assim, ainda não sabemos quais são as implicações práticas dessa abordagem.

O reuso de software tem sido um tópico de pesquisa há muitos anos; como explico nos capítulos 18 e 19, ainda permanecem muitas dificuldades práticas no reuso de software. Um dos principais problemas tem sido que os padrões para componentes reusáveis foram desenvolvidos somente recentemente e há muitos padrões em uso. Contudo, a iniciativa de Web services foi dirigida pelos padrões desde sua concepção, e os padrões que abrangem muitos aspectos de Web services ainda estão em desenvolvimento.

Figura 12.16

A arquitetura conceitual de um sistema orientado a serviços.

Os três padrões fundamentais que possibilitam comunicações entre Web services são:

1. *SOAP (*Simple Object Access Protocol*)*. Este protocolo define uma organização para troca estruturada de dados entre Web services.
2. *WSDL (*Web Services Description Language*)*. Este protocolo define como as interfaces dos Web services podem ser representadas.
3. *UDDI (*Universal Description, Discovery and Integration*)*. Este é um padrão de descobrimento que define como as informações de descrição do serviço, usadas pelos solicitantes do serviço para descobrir serviços, pode ser organizada.

Todos estes padrões são baseados em XML — uma linguagem de marcação legível por humanos e máquinas (Skonnard e Gudgin, 2002). Você não precisa, no entanto, conhecer detalhes desses padrões para compreender o conceito de Web services.

As arquiteturas de aplicação de Web services são arquiteturas não firmemente acopladas nas quais as ligações de serviços podem mudar durante a execução. Alguns sistemas serão somente construídos com a utilização de Web services e outros os integrarão com componentes desenvolvidos localmente. Para ilustrar como as aplicações podem ser organizadas, considere o seguinte cenário:

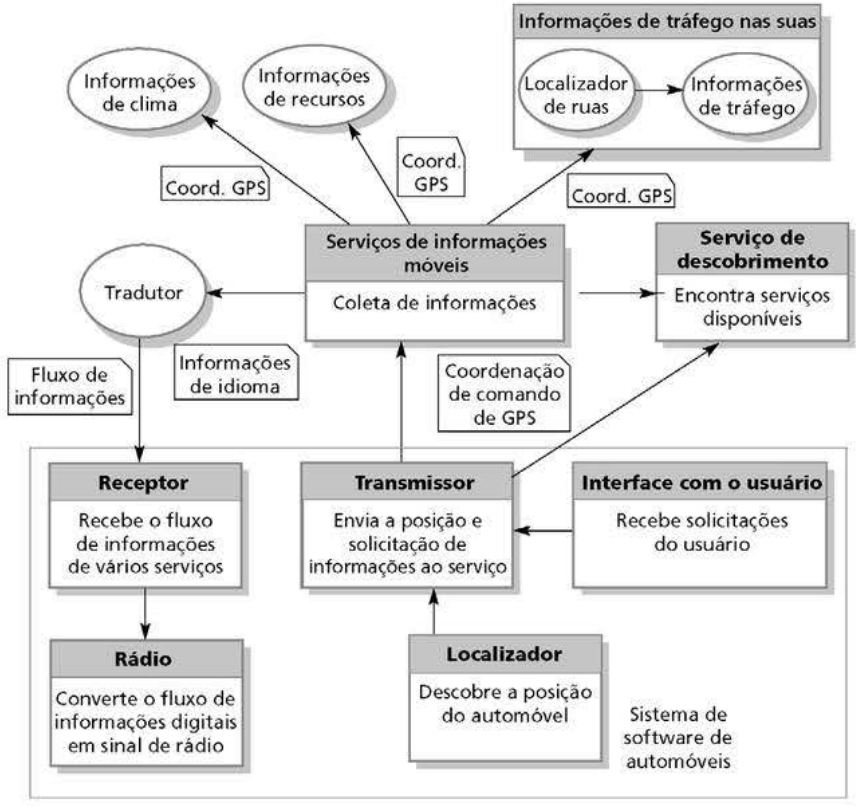
Um sistema de informações de um automóvel fornece aos motoristas informações sobre o clima, condições de tráfego, informações locais etc. O sistema é ligado ao aparelho de rádio do automóvel que apresenta as informações como um sinal de uma emissora de rádio específica. O automóvel é equipado com um receptor GPS para informar sua posição e, baseado nessa posição, o sistema acessa uma gama de serviços de informações. As informações podem ser fornecidas no idioma especificado pelo motorista.

A Figura 12.17 ilustra uma possível organização para tal sistema. O software do automóvel inclui cinco módulos. Estes cuidam da comunicação com o motorista, com o receptor GPS informando a posição do automóvel e com o aparelho de rádio do automóvel. Os módulos **Transmissor** e **Receptor** cuidam de todas as comunicações com os serviços externos.

O automóvel se comunica com um serviço de informações móvel fornecido externamente que agrega informações de uma gama de outros serviços que fornecem informações sobre o clima, informações sobre o tráfego e recursos locais. Provedores diferentes em localidades diferentes fornecem esses serviços e o sistema do automóvel usa um serviço de descobrimento para localizar o serviço de informações apropriado e se conectar a ele. O serviço de descobrimento é também usado pelo serviço de informações móvel para vincular os serviços apropriados de clima, tráfego e recursos. Os serviços trocam mensagens SOAP que

Figura 12.17

Sistema de informações de automóvel baseado em serviços.



incluem as informações de posição de GPS usadas pelos serviços para selecionar as informações adequadas. As informações agregadas retornam ao automóvel por meio de um serviço que traduz o idioma das informações no idioma do motorista.

Esse exemplo ilustra uma das principais vantagens da abordagem orientada a serviços. Não é necessário decidir quando o sistema será programado ou implantado, qual provedor de serviço deve ser usado e quais serviços específicos podem ser acessados. Quando o automóvel se move, o software interno usa o serviço de descobrimento para encontrar o serviço de informações mais apropriado e se conecta a ele. Devido ao uso de um serviço de tradução, o automóvel pode se mover além das fronteiras do país e, portanto, tornar as informações locais disponíveis para pessoas que não falam o idioma local.

Essa visão de computação orientada a serviços não é possível com os Web services atuais, nos quais a ligação de serviços com as aplicações é ainda praticamente estática. No entanto, no futuro, veremos mais ligações dinâmicas, arquiteturas de aplicação e concretização de sistemas dinâmicos orientados a serviços. Devido à importância desse tópico, ele será abordado mais detalhadamente no Capítulo 31, na nova seção sobre Tecnologias Emergentes.

PONTOS-CHAVE

- Os sistemas distribuídos podem suportar compartilhamento de recursos, abertura, concorrência, escalabilidade, tolerância a defeitos e transparência.
- Os sistemas cliente–servidor são sistemas distribuídos nos quais o sistema é modelado como um conjunto de serviços fornecidos por servidores para processos clientes.
- Em um sistema cliente–servidor, a interface com o usuário sempre funciona em um cliente, e o gerenciamento de dados é sempre fornecido por um servidor compartilhado. A funcionalidade da aplicação pode ser implementada no computador cliente ou em um servidor.
- Em uma arquitetura de objetos distribuídos, não há distinção entre clientes e servidores. Os objetos fornecem serviços gerais que podem ser chamados a partir de outros objetos. Essa abordagem pode ser usada para implementação de sistemas cliente–servidor.
- Os sistemas de objetos distribuídos requerem um middleware para cuidar das comunicações entre objetos e permitir que os objetos sejam adicionados e removidos do sistema.
- Os padrões CORBA são um conjunto de padrões para middleware que suporta arquiteturas de objetos distribuídos. Eles incluem definições de modelos de objetos, definições de um requisitor de objetos e definições de serviços comuns. Várias implementações de padrões CORBA estão disponíveis.
- As arquiteturas ponto a ponto são arquiteturas descentralizadas nas quais não há clientes e servidores distintos. As computações podem ser distribuídas em muitos sistemas em organizações diferentes.
- Sistemas orientados a serviços são criados pela ligação de serviços de software fornecidos por vários fornecedores de serviços. Um importante aspecto de arquiteturas orientadas a serviços é que a ligação de serviços aos componentes de arquitetura pode ser postergada até que o sistema seja implantado ou esteja em execução.

LEITURAS SUGERIDAS

"Turning software into a service". Artigo com boa visão geral que explica os princípios da computação orientada a serviços. Diferentemente de muitos artigos sobre esse assunto, ele não oculta os princípios com uma explicação sobre os padrões envolvidos. (M. Turner, et al., *IEEE computer*, 36 (10), out. 2003.)

Peer-to-peer: harnessing the power of disruptive technologies. Embora este livro não contenha uma grande quantidade de arquiteturas p2p, é uma excelente introdução à computação p2p e explica a organização e a abordagem usada em uma série de sistemas p2p. (A. Oram (ed.), 2001, O'Reilly e Associates, Inc.)

Distributed systems: concepts and design, 3rd ed. Livro abrangente que explica todos os aspectos de projeto e implementação de sistemas distribuídos. Os primeiros dois capítulos são particularmente importantes. (G. Couloris, et al., 2001, Addison-Wesley.)

"Middleware: a model for distributed systems services". É um excelente artigo com uma visão geral que resume o papel do middleware em sistemas distribuídos e explica a gama de serviços de middleware que podem ser fornecidos. (P. A. Bernstein, *Comm. ACM*, 39 (2), fev. 1996.)

EXERCÍCIOS

12.1 Explique por que os sistemas distribuídos são inerentemente mais escalónáveis do que sistemas centralizados. Quais são os prováveis limites para a escalabilidade do sistema?

12.2 Qual é a diferença fundamental entre uma abordagem cliente-gordo e cliente-magro para o desenvolvimento de sistemas cliente–servidor? Explique por que o uso de Java como implementação de linguagem confunde a distinção entre essas abordagens.

12.3 Seu cliente quer desenvolver um sistema para informações sobre ações no qual os interessados possam acessar informações sobre empresas e avaliar diversos cenários de investimentos usando um sistema de simulação. Cada interessado usa essa simulação de maneira diferente, de acordo com sua experiência e o tipo de ações. Sugira uma arquitetura cliente–servidor para esse sistema que mostre onde a funcionalidade está localizada. Justifique o modelo de sistema cliente–servidor escolhido.

12.4 Fazendo referência ao modelo de aplicação mostrado na Figura 12.4, explique os problemas que podem surgir na conversão de um sistema legado de processamento de apólices de seguro da década de 1980 em uma arquitetura cliente–servidor.

12.5 Quais são os recursos básicos que devem ser fornecidos por um requisitor de objetos?

12.6 Explique por que o uso de objetos distribuídos com um requisitor de objetos simplifica a implementação de sistemas cliente–servidor escalonáveis. Ilustre sua resposta com um exemplo.

12.7 Como é usado o CORBA IDL para apoiar as comunicações entre objetos implementados em diferentes linguagens de programação? Explique por que essa abordagem pode causar problemas de desempenho se houver diferenças radicais entre as linguagens usadas para a implementação de objetos.

12.8 Usando uma abordagem de objetos distribuídos, proponha uma arquitetura para um sistema de reservas de teatro no qual os usuários possam verificar a disponibilidade de lugares e reservá-los em um grupo de teatros. O sistema deve suportar a devolução de ingressos de modo que as pessoas possam devolver seus ingressos para revenda a outros clientes no último minuto.

12.9 Forneça duas vantagens e duas desvantagens de arquiteturas ponto a ponto descentralizadas e semicentralizadas.

12.10 Quais são as vantagens da ligação dinâmica em um sistema orientado a serviços?

12.11 Para um sistema de informações de automóvel, explique por que é melhor que o software do automóvel se comunique com um serviço de agregação em vez de se comunicar diretamente com os serviços de informações. Você deve considerar problemas como confiabilidade da comunicação na formulação de sua resposta.

12.12 O desenvolvimento da computação orientada a serviços foi baseado em especificações e na adoção de padrões recentes. Explique o papel geral da padronização no apoio e na restrição da competição e inovação no mercado de software.

13

Arquiteturas de aplicações

Objetivos

O objetivo deste capítulo é apresentar modelos de arquitetura para classes específicas de sistemas de software de aplicações. Após ler este capítulo, você:

- estará informado sobre duas organizações fundamentais de arquitetura de sistemas de negócios, chamadas *processamento em lotes* e *por transações*;
- compreenderá a arquitetura abstrata de informações e os sistemas de gerenciamento de recursos;
- compreenderá como sistemas dirigidos a comandos, como editores, podem ser estruturados como sistemas de processamento de eventos;
- conhecerá a estrutura e a organização de sistemas de processamento de linguagens.

Conteúdo

13.1 Sistemas de processamento de dados
13.2 Sistemas de processamento de transações
13.3 Sistemas de processamento de eventos
13.4 Sistemas de processamento de linguagens

Conforme explicado no Capítulo 11, você pode olhar as arquiteturas de sistemas de várias maneiras. Assim, as explicações sobre arquiteturas de sistemas nos capítulos 11 e 12 se concentraram em perspectivas de arquitetura e tópicos tais como controle, distribuição e estruturação de sistema. Neste capítulo, contudo, explico uma abordagem alternativa e enxergo as arquiteturas sob a perspectiva das aplicações.

Sistemas de aplicações são criados para atender algumas necessidades de negócio ou organizacionais. Todos os negócios têm muito em comum — eles necessitam contratar pessoas, emitir faturas, administrar as contas etc. — e isso é especialmente verdadeiro no caso dos negócios que operam no mesmo setor. Portanto, assim como as funções gerais de negócio, todas as empresas telefônicas necessitam de sistemas para conectar chamadas, gerenciar sua rede, emitir contas para os clientes etc. Conseqüentemente, os sistemas de aplicações que esses negócios usam também têm muito em comum.

Geralmente, os sistemas de mesmo tipo possuem arquiteturas similares, e as diferenças entre esses sistemas estão nos detalhes da funcionalidade fornecida. Isso pode ser ilustrado pelo crescimento dos chamados sistemas de Enterprise Resource Planning (ERP), como o sistema SAP/R3 (Appelrath e Ritter, 2000) e pacotes de software verticais para aplicações específicas. Nesses sistemas, discutidos no Capítulo 18, um sistema genérico é configurado e adaptado para criar uma

aplicação específica de negócios. Por exemplo, um sistema para gerenciamento de rede de fornecedores pode ser adaptado para diferentes tipos de fornecedores, produtos e contratos.

Na explicação sobre arquiteturas de aplicações deste capítulo, apresento modelos genéricos de estrutura de vários tipos de aplicações. Discuto a organização básica desses tipos de aplicações e, quando apropriado, detalho a arquitetura de alto nível para mostrar os subsistemas normalmente incluídos nas aplicações.

Como um projetista de software, você pode usar arquiteturas genéricas de aplicações de diversas maneiras:

1. *Como um ponto de partida para o processo de projeto de arquitetura*. Se você não está familiarizado com esse tipo de aplicação, baseie seus projetos iniciais em arquiteturas genéricas. Naturalmente, estes deverão ser próprios para sistemas específicos, mas são um bom ponto de partida para seu projeto.
2. *Como um checklist do projeto*. Se você desenvolveu um projeto de arquitetura de sistema, é possível compará-lo com a arquitetura genérica de aplicações para checar se falta considerar algum componente importante de projeto.
3. *Como uma maneira de organização do trabalho da equipe de desenvolvimento*. As arquiteturas de aplicações identificam características estruturais estáveis das arquiteturas de sistema e, em muitos casos, é possível desenvolvê-las paralelamente. Você pode atribuir atividades para os membros de projeto para implementar subsistemas diferentes dentro da arquitetura.
4. *Como meio de avaliação de componentes para reuso*. Se você tem componentes que podem ser reusados, compare-os com estruturas genéricas para verificar se o reuso é possível na aplicação em desenvolvimento.
5. *Como vocabulário para conversar sobre tipos de aplicações*. Se você estiver discutindo sobre uma aplicação específica ou tentando comparar aplicações do mesmo tipo, poderá usar os conceitos identificados na arquitetura genérica para falar sobre as aplicações.

Existem muitos tipos de sistema de aplicações e, genericamente, eles podem parecer muito diferentes. Contudo, quando você examina a organização de arquitetura das aplicações, muitas dessas aplicações aparentemente diferentes têm muito em comum. Demonstro isso neste capítulo ao descrever arquiteturas de quatro tipos abrangentes de aplicações:

1. *Aplicações de processamento de dados*. São aplicações voltadas a dados. Elas processam dados em lotes sem intervenções explícitas do usuário durante o processamento. As ações específicas tomadas pela aplicação dependem dos dados que são processados. Os sistemas de processamento em lotes são normalmente usados em aplicações de negócios nas quais as operações similares são realizadas sobre uma grande quantidade de dados. Eles tratam de uma grande variedade de funções administrativas, como folha de pagamento, cobrança, contabilidade e publicidade.
2. *Aplicações de processamento de transações*. São aplicações centradas em banco de dados que processam as solicitações de informações provenientes de usuários e que atualizam as informações no banco de dados. Esse é o tipo mais comum de sistemas de negócios interativos. Elas são organizadas de tal maneira que as ações do usuário não podem interferir umas nas outras e a integridade do banco de dados é mantida. Essa classe de sistema inclui sistemas bancários interativos, sistemas de e-commerce, sistemas de informações e sistemas de reservas.
3. *Sistemas de processamento de eventos*. Esta é uma classe muito grande de aplicações na qual as ações de sistema dependem da interpretação de eventos no ambiente do sistema. Esses eventos podem ser a entrada de um comando do usuário ou a mudança de variáveis monitoradas pelo sistema. Muitas aplicações baseadas em PC, incluindo jogos, sistemas de edição tais como processadores de texto, planilhas, editores de imagens e sistemas de apresentação são sistemas de processamento de eventos. Sistemas em tempo real, explicados no Capítulo 15, também pertencem a esta categoria.
4. *Sistemas de processamento de linguagens*. São sistemas em que as intenções do usuário são expressas em uma linguagem formal (como Java). O sistema de processamento de linguagens processa a linguagem em um formato interno e, em seguida, interpreta a representação. Os sistemas de processamento de linguagens bem conhecidos são compiladores que traduzem programas em linguagem de alto nível para código de máquina. No entanto, esses sistemas também são usados para interpretar linguagens de comandos para banco de dados, sistemas de informações e linguagens de marcação, como XML (Harold e Means, 2002), extensivamente usada para descrever itens de dados estruturados.

Escolhi esses tipos de sistema específicos pelo fato de eles representarem a maioria dos sistemas em uso atualmente. Sistemas de negócios são em geral sistemas de processamento de dados ou transações, e a maioria dos softwares de computadores pessoais é construída em torno de uma arquitetura de processamento de eventos. Sistemas de tempo real também são sistemas de processamento de eventos; abordo essas arquiteturas no Capítulo 15. Todo o desenvolvimento de software conta com sistemas de processamento de linguagem, como os compiladores.

Tanto os sistemas de processamento de lotes como os de transações são centrados em banco de dados. Por causa da importância central dos dados, é comum que aplicações de tipos diferentes compartilhem o mesmo banco de dados. Por exemplo, um sistema de processamento de dados de negócios que imprime extratos bancários usa o mesmo banco de dados de contas de clientes como um sistema de processamento de transações que fornece acesso baseado na Web para as informações de conta.

Naturalmente, conforme discuti no Capítulo 11, aplicações complexas pouquíssimas vezes seguem um modelo de arquitetura única e simples. Em vez disso, sua arquitetura é muitas vezes híbrida, com diferentes partes da aplicação estruturadas de maneiras diferentes. Portanto, ao projetar esses sistemas, você precisa considerar as arquiteturas de subsistemas individuais e como essas partes são integradas dentro de uma arquitetura global de sistema.

13.1 Sistemas de processamento de dados

As empresas contam com sistemas de processamento de dados que apóiam muitos aspectos de negócios, como pagamento de salários, cálculo e impressão de faturas, manutenção de contas e emissão de renovações de apólices de seguro. Como o nome indica, esses sistemas enfocam os dados e, geralmente, os bancos de dados são várias vezes maiores que os sistemas em si. Os sistemas de processamento de dados são sistemas de processamento em lotes nos quais os dados são entradas e saídas em lotes de um arquivo ou banco de dados em vez de entradas ou saídas para um terminal de usuário. Esses sistemas selecionam os dados de registros de entrada e, dependendo do valor dos campos nos registros, tomam algumas ações especificadas no programa. Eles podem, então, enviar o resultado novamente do processamento ao banco de dados e formatar a entrada e a saída processada para impressão.

A arquitetura dos sistemas de processamento de lotes tem três componentes principais, conforme ilustrados na Figura 13.1. Um componente de entrada coleta as entradas de uma ou mais fontes, um componente de processamento realiza a computação usando essas entradas e um componente de saída gera saídas a serem escritas novamente no banco de dados e impressas. Por exemplo, um sistema de cobrança de contas de telefone obtém os registros do cliente e as leituras do medidor de telefone (entradas) de uma chave comutadora, calcula os custos para cada cliente (processo) e, em seguida, imprime as contas (saídas) para cada cliente.

Os componentes de entrada, de processamento e de saída podem ser decompostos ainda em uma estrutura entrada-processo-saída. Por exemplo:

1. Um componente de entrada pode ler algum dado (entrada) de um arquivo ou banco de dados, verificar a validade dos dados e corrigir alguns erros (processo) e, depois enfileirar os dados válidos para processamento (saída).
2. Um componente de processamento pode obter uma transação de uma fila (entrada), realizar algumas computações que envolvam os dados e criar um novo registro de dados que registre os resultados da computação (processo); depois enfileirar esse novo registro para impressão (saída). Algumas vezes o processamento é feito dentro do banco de dados do sistema e algumas vezes é um programa separado.
3. Um componente de saída pode ler registros de uma fila (entrada), formatar esses registros de acordo com o formulário de saída (processo) e enviá-los para uma impressora ou escrever os novos registros novamente no banco de dados (saída).

A natureza dos sistemas de processamento de dados, nos quais os registros e as transações são processados em série, sem a necessidade de manter o estado entre as transações, significa que esses sistemas são naturalmente orientados a funções, em vez de orientados a objetos. As funções são componentes que não mantêm informações de estado interno de uma chamada

Figura 13.1

Modelo de entrada–processo–saída de um sistema de processamento de dados.

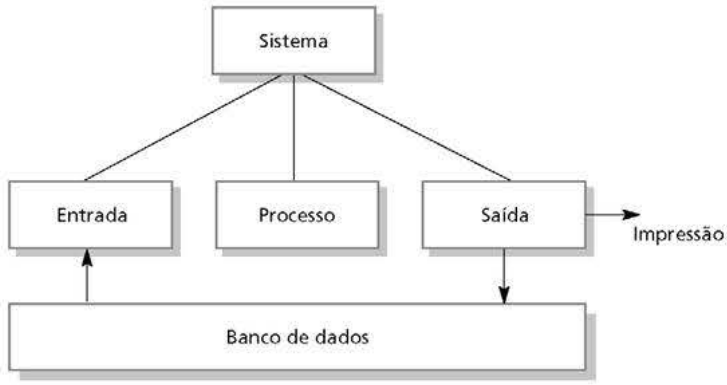


Figura 13.2 Diagrama de fluxo de dados de um sistema de folha de pagamentos.

para outra. Os diagramas de fluxo de dados, apresentados no Capítulo 8, são uma boa maneira de descrever a arquitetura de sistemas de processamento de dados de negócios.

Os diagramas de fluxo de dados são uma maneira de representar sistemas orientados a funções, em que cada retângulo de bordas arredondadas no fluxo de dados corresponde a uma função que implementa alguma transformação de dados, e cada seta corresponde a um item de dados processado pela função. Arquivos ou depósitos de dados são representados por retângulos. A vantagem de diagramas de fluxo de dados é que eles demonstram o processamento de ponta a ponta. Isto é, você pode ver todas as funções que atuam sobre os dados quando eles se movem através dos estágios do sistema. A estrutura fundamental do fluxo de dados consiste de uma função de entrada que passa os dados para uma função de processamento e, então, para uma função de saída.

A Figura 13.2 ilustra como os diagramas de fluxo de dados podem ser usados para demonstrar uma visão mais detalhada da arquitetura de um sistema de processamento de dados. Essa figura mostra o projeto de um sistema de pagamento de salários. Nesse sistema, as informações sobre os empregados da organização são lidas, os salários e as deduções são calculados mensalmente e os pagamentos são efetuados. Você pode ver como esse sistema segue uma estrutura básica de entrada–processo–saída:

1. As funções, à esquerda do diagrama, **Ler registro do empregado**, **Ler dados de pagamento mensalmente** e **Validar dados do empregado** inserem os dados de cada empregado e verificam esses dados.
2. A função **Calcular salário** calcula o salário total bruto de cada empregado e as várias deduções do salário. Assim, o salário líquido mensal é calculado.
3. As funções de saída escrevem uma série de arquivos que mantêm detalhes das deduções efetuadas no salário a ser pago. Esses arquivos são processados por outros programas ao mesmo tempo que os detalhes de todos os empregados são calculados. Um demonstrativo de pagamento de empregado, que registra o pagamento líquido e as deduções efetuadas, é impresso pelo sistema.

O modelo de arquitetura dos programas de processamento de dados é relativamente simples. Contudo, nesses sistemas, a complexidade da aplicação muitas vezes reflete-se nos dados que são processados. O projeto de arquitetura de sistema, portanto, envolve raciocinar sobre arquitetura de dados (Bracket, 1994), assim como sobre a arquitetura do programa. O projeto de estruturas de dados está fora do escopo deste livro.

13.2 Sistemas de processamento de transações

Os sistemas de processamento de transações são projetados para processar solicitações de informações por usuários de um banco de dados ou solicitações para atualizar o banco de dados (Lewis, et al., 2003). Tecnicamente, uma

transação de banco de dados é uma seqüência de operações tratada como uma unidade simples (uma unidade atômica). Todas as operações de uma transação precisam ser realizadas antes que as mudanças tornem-se permanentes no banco de dados. Isso significa que falhas de operações dentro da transação não conduzem a inconsistências no banco de dados.

Um exemplo de transação é uma solicitação de um cliente para retirar dinheiro de sua conta bancária usando um caixa eletrônico. Isso envolve a obtenção de detalhes da conta do cliente, verificação do saldo, modificação do saldo pela quantidade retirada e envio de comandos para o caixa eletrônico para liberação do dinheiro. Até que todas essas etapas sejam concluídas, a transação estará incompleta e o banco de dados de contas de clientes não será alterado.

Da perspectiva do usuário, uma transação é qualquer seqüência coerente de operações que satisfaz um objetivo, tal como 'encontrar os vôos programados de Londres a Paris'. Se a transação do usuário não exigir que o banco de dados seja alterado, pode ser necessário considerá-la como uma transação técnica de banco de dados.

Os sistemas de processamento de transações são geralmente sistemas interativos nos quais os usuários enviam solicitações assíncronas de serviço. A Figura 13.3 ilustra a estrutura de arquitetura em alto nível dessas aplicações. Primeiro, um usuário faz uma solicitação para o sistema através de um componente de processamento de E/S. A solicitação é processada por alguma lógica específica da aplicação. Uma transação é criada e passada para um gerenciador de transações, que é em geral incorporado ao sistema de gerenciamento de banco de dados. Depois que o gerenciador de transações assegurar que a transação foi concluída adequadamente, ele sinalizará para a aplicação que o processamento foi finalizado.

A estrutura entrada-processo-saída que podemos ver nas aplicações de processamento de dados também se aplica aos muitos sistemas de processamento de transações. Alguns desses sistemas são versões interativas de sistemas de processamento de lotes. Por exemplo, antes os bancos enviavam todas as transações dos clientes de modo off-line e, então, toda noite executavam essas transações em um lote no banco de dados de contas. Essa abordagem foi, na maioria das vezes, substituída por sistemas interativos e baseados em transações que atualizam as contas em tempo real.

Um exemplo de sistema de processamento de transações é um sistema bancário que permite aos clientes consultarem suas contas e retirarem dinheiro de um caixa eletrônico. O sistema é composto de dois subsistemas de software que cooperam entre si — o software de caixa eletrônico e o software de processamento de contas localizadas no servidor de banco de dados. Os subsistemas de entrada e de saída são implementados como softwares no caixa eletrônico, uma vez que o subsistema de processamento está no servidor de banco de dados. A Figura 13.4 mostra a arquitetura desse sistema. Adicionei alguns detalhes no diagrama básico de entrada-processo-saída para mostrar os componentes que podem ser envolvidos nas atividades de entrada, de processamento e de saída. Deliberadamente, não sugeri como esses componentes interagem, pois a seqüência de operação pode diferir de uma máquina para outra.

Em sistemas como os de contabilidade de clientes de um banco, pode haver diferentes maneiras de interagir com o sistema. Muitos clientes interagirão por meio de caixas eletrônicos, mas uma equipe do banco usará terminais de mesa para acessar o sistema. Pode haver vários tipos de caixas eletrônicos e terminais de mesa, e alguns clientes e a equipe do banco podem acessar os dados de contas por meio de navegadores Web.

Para simplificar o gerenciamento de diferentes protocolos de comunicação entre terminais, sistemas de processamento de transações de larga escala podem incluir middleware para comunicação com todos os tipos de terminal, organização e ordenação em série dos dados dos terminais e envio dos dados para processamento. Esse middleware, explicado brevemente

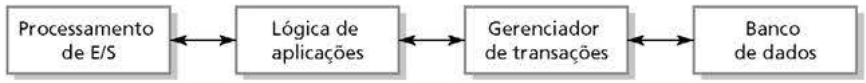


Figura 13.3

Estrutura de aplicações de processamento de transações.

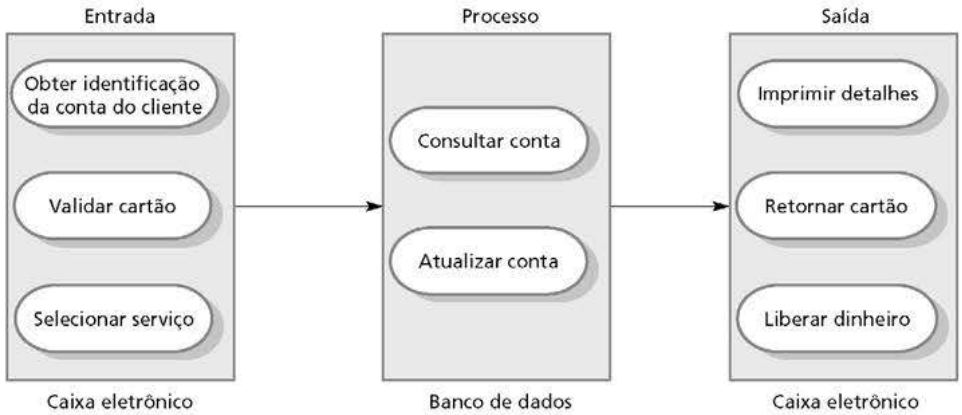


Figura 13.4

Arquitetura de software de um caixa eletrônico.

no Capítulo 12, pode ser chamado de monitor de teleprocessamento ou sistema de gerenciamento de transações. O CICS da IBM (Horswill e Miller, 2000) é um bom exemplo desse tipo de sistema amplamente usado.

A Figura 13.5 mostra outra visão de arquitetura de um sistema de contabilidade de clientes que cuida das transações de contas pessoais de caixas eletrônicos e terminais de mesa em um banco. O monitor de teleprocessamento cuida das entradas e ordena as transações em série, as quais são convertidas em consultas de banco de dados. O processamento de consultas ocorre no sistema de gerenciamento de banco de dados. Os resultados são retornados para o monitor de teleprocessamento, que mantém o acompanhamento dos terminais que fazem as solicitações. Depois o sistema organiza os dados em um formulário que pode ser tratado pelo software do terminal e retorna os resultados da transação para ele.

13.2.1 Sistemas de gerenciamento de informações e recursos

Todos os sistemas que envolvem interação com um banco de dados compartilhado podem ser considerados como sistemas de informações baseados em transações. Um sistema de informações permite acesso controlado de uma grande base de informações, tais como catálogo de bibliotecas, tabela de horários de vôos ou registros de pacientes em um hospital. O desenvolvimento da Web fez com que um grande número de sistemas de informações migrasse de sistemas organizacionais especializados para sistemas de propósito geral acessíveis universalmente.

A Figura 13.6 é um modelo muito geral de um sistema de informações. O sistema é modelado com o uso de uma abordagem de camadas ou de máquina abstrata (explicada na Seção 11.2.3) na qual a camada superior apóia a interface com o usuário e a camada inferior, o banco de dados de sistema. A camada de comunicações de usuário cuida de todas as entradas e saídas de interface com o usuário, e a camada de recuperação de informações inclui uma lógica específica de aplicação para acesso e atualização do banco de dados. Como veremos mais adiante, as camadas neste modelo podem ser mapeadas diretamente em servidores em um sistema baseado na Internet.

Como exemplo de uma instância desse modelo de camadas, a Figura 13.7 apresenta a arquitetura do sistema LIBSYS. Lembre-se de que esse sistema permite aos usuários acessar documentos em bibliotecas remotas e baixar documentos para impressão. Adicionei detalhes de cada camada no modelo identificando os componentes que apóiam as comunicações de usuários e a recuperação e o acesso às informações. Você deve também notar que o banco de dados é distribuído. Os usuários se conectam realmente, pelo sistema, aos bancos de dados das bibliotecas que fornecem documentos.

A camada de comunicação de usuário na Figura 13.7 inclui três componentes principais:

Figura 13.5

Middleware para gerenciamento de transações.

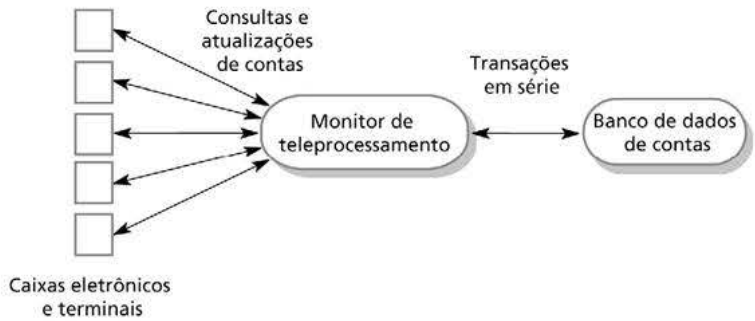


Figura 13.6

Modelo em camadas de um sistema de informações.

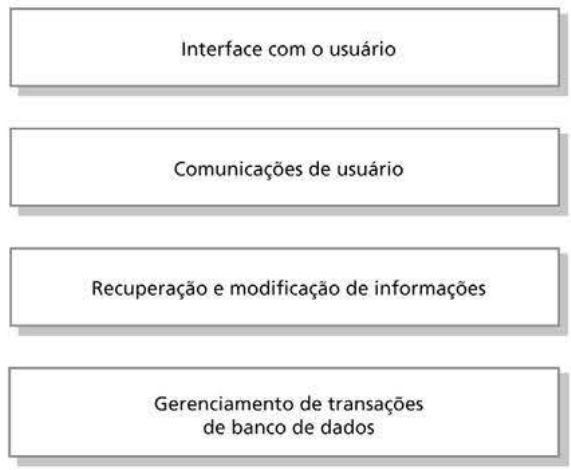
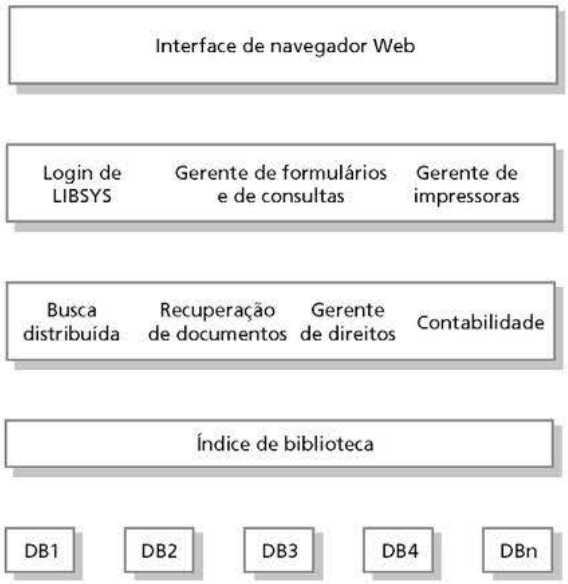


Figura 13.7

A arquitetura do sistema LIBSYS.



1. *O componente de login do LIBSYS* identifica e autentica os usuários. Todos os sistemas de informações com acesso restrito a um grupo de usuários conhecidos precisam ter a autenticação de usuário como parte fundamental do sistema de comunicação de usuário. A autenticação de usuário pode ser pessoal, mas em sistemas de e-commerce pode também exigir o fornecimento de detalhes de cartões de crédito.
2. *O componente gerenciador de formulários e de consultas* gerencia os formulários que podem ser apresentados aos usuários e fornece recursos de consulta que permitem ao usuário solicitar informações do sistema. Novamente, todos os sistemas de informações devem incluir um componente com esses recursos.
3. *O componente gerenciador de impressoras* é específico do LIBSYS. Ele controla a impressão de documentos que, por razões de direitos autorais, pode ser limitada. Por exemplo, alguns documentos podem ser somente impressos uma vez em impressoras da biblioteca registrada.

A camada de recuperação e modificação de informações do sistema LIBSYS inclui componentes específicos de aplicação que implementam a funcionalidade do sistema. Esses componentes são:

1. *Busca distribuída*. Este componente procura documentos em resposta às consultas de usuários em todas as bibliotecas registradas no sistema. A lista de bibliotecas conhecidas é mantida no índice de bibliotecas.
2. *Recuperação de documentos*. Este componente recupera o documento ou documentos solicitados pelo usuário do servidor onde o sistema LIBSYS está sendo executado.
3. *Gerenciador de direitos*. Este componente trata de todos os aspectos de gerenciamento de direitos digitais e de direitos autorais. Ele mantém registros de quem requisitou documentos e, por exemplo, assegura que não possam ser feitas várias solicitações para o mesmo documento pela mesma pessoa.
4. *Contabilidade*. Este componente registra todas as solicitações e, se necessário, cuida de quaisquer encargos criados pelas bibliotecas no sistema. Ele também produz relatórios gerenciais de utilização do sistema.

Podemos ver a mesma estrutura de quatro camadas em outro tipo de sistema de informações, especificamente sistemas projetados para apoiar a alocação de recursos. Os sistemas de alocação de recursos gerenciam uma quantidade fixa de algum recurso definido, tais como ingressos para um concerto ou jogo de futebol. Esses ingressos são alocados aos usuários que solicitaram esse recurso de um fornecedor. Sistemas de emissão de ingressos são um exemplo óbvio de sistema de alocação de recursos, mas uma grande quantidade de programas aparentemente diferentes também são sistemas de alocação de recursos. Alguns exemplos dessa classe de sistema são:

1. *Sistemas de tabelas de horários* que alocam períodos em espaços de tabelas de horários. Esse recurso alocado constitui um período de tempo, e ele geralmente impõe muitas restrições associadas a cada demanda.
2. *Sistemas de bibliotecas* que gerenciam o empréstimo e a retirada de livros ou outros itens. Neste caso, os recursos que são alocados são itens que podem ser solicitados para empréstimo. Neste tipo de sistema, os recursos não são simplesmente alocados, mas devem algumas vezes ser desalocados do usuário do recurso.

3. *Sistemas de gerenciamento de tráfego aéreo* nos quais o recurso que é alocado é um segmento do espaço aéreo, de modo que a separação é mantida entre duas aeronaves que são gerenciadas pelo sistema. Novamente, isso envolve alocação e realocação dinâmicas do recurso, mas este é um recurso virtual e não físico.

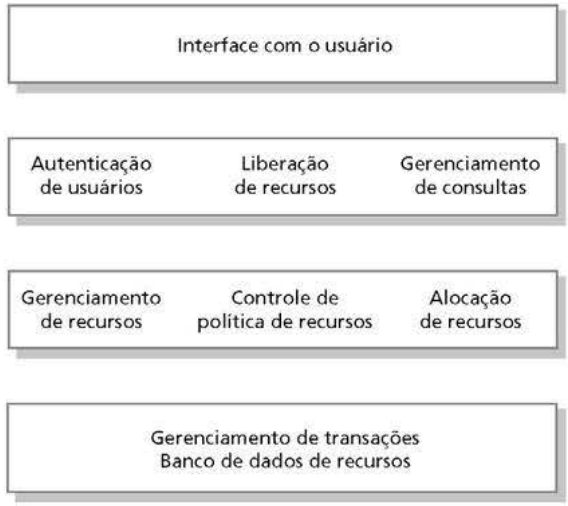
Sistemas de alocação de recursos são uma classe de aplicação amplamente usada. Se olharmos sua arquitetura detalhadamente, poderemos ver como ela está alinhada com o modelo de sistema de informações mostrado na Figura 13.6. Os componentes de um sistema de alocação de recursos (mostrado na Figura 13.8) inclui:

1. Um *banco de dados de recursos* que mantém detalhes de recursos que são alocados. Os recursos podem ser adicionados ou removidos do banco de dados. Por exemplo, em um sistema de bibliotecas, o banco de dados de recursos inclui detalhes de todos os itens que podem ser solicitados para empréstimo pelos usuários da biblioteca. Normalmente, isso é implementado por meio de um sistema de gerenciamento de banco de dados que inclui um sistema de processamento de transações. O sistema de gerenciamento de banco de dados também inclui funcionalidades de bloqueio de recursos de tal maneira que o mesmo recurso não possa ser alocado a usuários que fizeram solicitações simultâneas.
2. Um *conjunto de regras* que descreve as regras de alocação de recursos. Por exemplo, um sistema de biblioteca normalmente limita quem pode ser alocado a um recurso (usuários de biblioteca registrados), o período em que um livro ou outro item pode ser solicitado para empréstimo, o número máximo de livros que pode ser solicitado para empréstimo etc. Isso é encapsulado no componente de controle de política de recursos.
3. Um *componente de gerenciamento de recursos* que permite que o provedor de recursos adicione, edite ou elimine recursos do sistema.
4. Um *componente de alocação de recursos* que atualiza o banco de dados de recursos quando eles são designados e que associa esses recursos a detalhes do solicitante do recurso.
5. Um *módulo de autenticação de usuários* que permite ao sistema verificar se os recursos estão sendo alocados para um usuário reconhecido. Em um sistema de biblioteca, isso pode ser um cartão de biblioteca lido por uma máquina; em um sistema de alocação de ingressos, pode ser um cartão de crédito que verifica se o usuário está capacitado a pagar pelo recurso.
6. Um *módulo de gerenciamento de consultas* que permite ao usuário descobrir quais recursos estão disponíveis. Em sistema de biblioteca, isso seria tipicamente baseado em consultas de determinados itens; em um sistema de emissão de ingressos, poderia envolver um display gráfico que mostrasse quais ingressos estão disponíveis para datas específicas.
7. Um *componente de entrega de recursos* que prepara os recursos a serem entregues ao solicitante. Em um sistema de emissão de ingressos, isso pode envolver a preparação de uma confirmação por e-mail, o envio de uma solicitação para uma impressora que imprime os ingressos e os detalhes de para onde os ingressos devem ser enviados.
8. Um *componente de interface com o usuário* (freqüentemente um navegador Web) que está fora do sistema e permite ao solicitante do recurso enviar consultas e solicitações para o recurso a ser alocado.

Esta arquitetura de camadas pode ser realizada de várias maneiras. O software de sistemas de informações pode ser organizado de modo que cada camada seja um componente de larga escala em execução em um servidor separado. Cada camada

Figura 13.8

Modelo em camadas de um sistema de alocação de recursos.



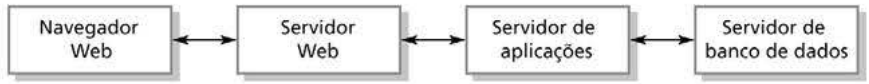


Figura 13.9
Sistema de processamento de transações de Internet de várias camadas.

define suas interfaces externas e toda comunicação ocorre através dessas interfaces. Alternativamente, se todo o sistema de informações for executado em um único computador, geralmente as camadas intermediárias serão implementadas como um único programa que se comunica com o banco de dados através de sua API. Uma terceira alternativa é implementar componentes de baixa granularidade como serviços Web separados (explicados no Capítulo 12), compondo-os dinamicamente de acordo com as solicitações dos usuários.

As implementações de sistemas de gerenciamento de informações e de recursos baseados em protocolos de Internet são atualmente uma norma; a interface com o usuário nesses sistemas é implementada por meio de um navegador Web. A organização de servidores nesses sistemas reflete o modelo de quatro camadas apresentado na Figura 13.6. Esses sistemas são geralmente implementados como arquiteturas cliente–servidor de várias camadas, conforme explicadas no Capítulo 12. A organização do sistema é mostrada na Figura 13.9. O servidor Web é responsável por todas as comunicações de usuário; o servidor de aplicações é responsável pela implementação da lógica específica de aplicação, assim como pelos pedidos de armazenamento e de recuperação de informações. O uso de vários servidores permite alta taxa de informações e torna possível cuidar de centenas de transações por minuto.

Os sistemas de e-commerce são sistemas de gerenciamento de recursos baseados em Internet projetados para aceitar pedidos eletrônicos de produtos ou serviços e, então, organizar a entrega desses produtos ou serviços para o cliente. Atualmente, existe uma grande gama desses sistemas em uso, variando de sistemas que permitem a organização de serviços como aluguel de automóveis a sistemas que apóiam pedidos de produtos tangíveis, como livros ou alimentos. Em um sistema de e-commerce, a camada específica de aplicação inclui uma funcionalidade adicional que apóia um 'carrinho de compras' no qual os usuários podem colocar uma quantidade de itens em transações separadas e depois pagá-los todos juntos em uma única transação.

13.3 Sistemas de processamento de eventos

Os sistemas de processamento de eventos respondem aos eventos do ambiente ou da interface com o usuário do sistema. Conforme expliquei no Capítulo 11, a principal característica dos sistemas de processamento de eventos é que a seqüência de eventos é imprevisível e o sistema deve ser capaz de trabalhar com esses eventos quando eles ocorrerem.

Todos nós usamos esses sistemas baseados em eventos em nossos próprios computadores — processadores de texto, sistemas de apresentação e jogos são todos dirigidos por eventos da interface com o usuário. O sistema detecta e interpreta eventos. Os eventos da interface com o usuário representam comandos implícitos para o sistema, que toma alguma ação para obedecer a esse comando. Por exemplo, se você está usando um processador de texto e clica duas vezes em uma palavra, esse clique duplo significa 'selecionar esta palavra'.

Os sistemas de tempo real, que executam uma ação em 'tempo real' em resposta a algum estímulo externo, são também sistemas de processamento baseados em eventos. Contudo, para sistemas de tempo real, os eventos não são geralmente eventos de interface com o usuário, mas eventos associados a sensores e atuadores do sistema. Devido à necessidade de resposta em tempo real a eventos imprevisíveis, esses sistemas de tempo real são normalmente organizados como um conjunto de processos que cooperam entre si. As arquiteturas genéricas para sistemas de tempo real serão abordadas no Capítulo 15.

Nesta seção, enfoco a descrição da arquitetura genérica de sistemas de edição. Os sistemas de edição são programas executados em PCs ou estações de trabalho e que permitem aos usuários editar documentos, como documentos de texto, diagramas ou imagens. Alguns editores concentram-se na edição de um único tipo de documento, como imagens de uma câmera digital ou um scaner. Outros, incluindo a maioria dos processadores de texto, são editores múltiplos e incluem apoio a diferentes tipos de edição, como textos e diagramas. Você pode até pensar em uma planilha eletrônica como um sistema de edição no qual você edita as células da planilha. Naturalmente, as planilhas contam com funcionalidade adicional para realizar cálculos.

Os sistemas de edição têm uma quantidade de características que os diferenciam de outros tipos de sistema e que influenciam seu projeto de arquitetura:

1. Os sistemas de edição são, na maioria, sistemas para um único usuário. Eles, portanto, não têm de lidar com problemas de concorrência múltipla de acesso aos dados e têm gerenciamento de dados mais simples do que os sistemas baseados em transações. Mesmo quando os dados são compartilhados, o gerenciamento de transações geralmente

não é usado porque as transações levam um longo tempo e métodos alternativos de manutenção de integridade de dados são usados.

2. Eles devem fornecer um rápido feedback das ações de usuário, como 'selecionar' e 'excluir'. Isso significa que eles precisam operar sobre representações de dados que são mantidos na memória do computador, e não no disco. Devido aos dados estarem em memória volátil, eles poderão ser perdidos se houver uma falha no sistema e, assim, os sistemas de edição devem estar prontos para uma recuperação de falhas.
3. As sessões de edição são geralmente muito mais demoradas do que sessões que envolvem pedidos de produtos, ou a realização de alguma outra transação. Isso significa, novamente, que há um risco maior de perda se surgirem problemas. Portanto, muitos sistemas de edição incluem recursos de recuperação que salvam automaticamente o trabalho em andamento e recuperam o trabalho para o usuário na ocorrência de uma falha do sistema.

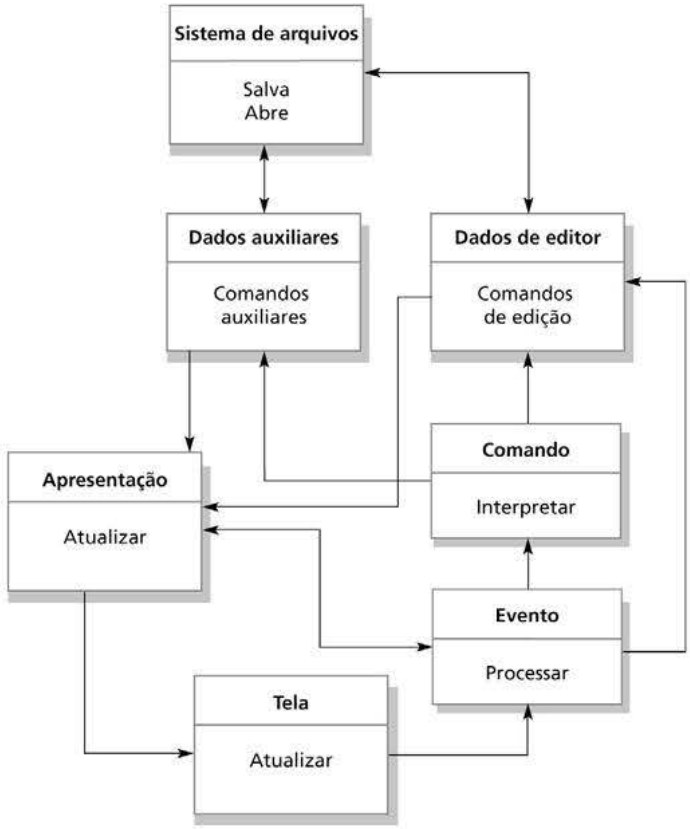
Uma arquitetura genérica de um sistema de edição é mostrada na Figura 13.10 como um conjunto de objetos que interagem. Os objetos do sistema são ativos, e não passivos (veja o Capítulo 14), e podem operar concorrente e autonomamente. Em sua essência, os eventos de tela são processados e interpretados como comandos. Os comandos atualizam a estrutura de dados que, então, é reapresentada na tela.

As responsabilidades dos componentes de arquitetura mostrados na Figura 13.10 são:

1. **Tela**. Esse objeto monitora o segmento de memória de tela e detecta os eventos que ocorrem. Esses eventos, depois, são passados para o objeto de processamento de eventos junto com suas coordenadas de tela.
2. **Evento**. Esse objeto é acionado por um evento que chega da **Tela**. Ele usa o conhecimento do que é apresentado para interpretar esse evento e para traduzi-lo em comandos de edição apropriados. Esse comando é, então, passado para o objeto responsável pela interpretação de comandos. Para eventos muito comuns, tais como cliques com o mouse ou acionamento de teclas no teclado, o objeto de eventos pode se comunicar diretamente com a estrutura de dados. Isso permite atualizações mais rápidas dessa estrutura.
3. **Comando**. Esse objeto processa um comando do objeto de eventos e chama o método apropriado no **objeto de dados Editor** para executar o comando.
4. **Dados de editor**. Quando o método de comando apropriado no **objeto de dados Editor** é chamado, ele atualiza a estrutura de dados e chama o método **Update** no **Display** para apresentar os dados modificados.

Figura 13.10

Modelo de arquitetura de um sistema de edição.



5. **Dados auxiliares**. Assim como a estrutura de dados em si, os editores gerenciam outros dados, como estilos e preferências. Neste modelo simples de arquitetura, agrupei isso em **dados auxiliares**. Alguns comandos de editor, como um comando para iniciar uma verificação de escrita, são implementados por um método nesse objeto.
6. **Sistema de arquivos**. Esse objeto cuida de todas as aberturas e salvamentos de arquivos. Os objetos podem estar em arquivos de dados de editor ou de dados auxiliares. Para evitar a perda de dados, muitos editores têm recursos de gravação automática que grava a estrutura de dados automaticamente. Isso pode, então, ser recuperado na ocorrência de uma falha do sistema.
7. **Apresentação**. Esse objeto mantém o acompanhamento da organização de apresentação de tela. Ele chama o método Refresh na Tela quando a apresentação é alterada.

Devido à necessidade de uma rápida resposta aos comandos do usuário, os sistemas de edição não possuem controlador central que chama os componentes que tomam ações. No lugar disso, os componentes críticos do sistema são executados concorrentemente e podem se comunicar diretamente (por exemplo, o processador de eventos pode se comunicar diretamente com a estrutura de dados do editor) de tal maneira que um desempenho maior pode ser atingido.

13.4 Sistemas de processamento de linguagens

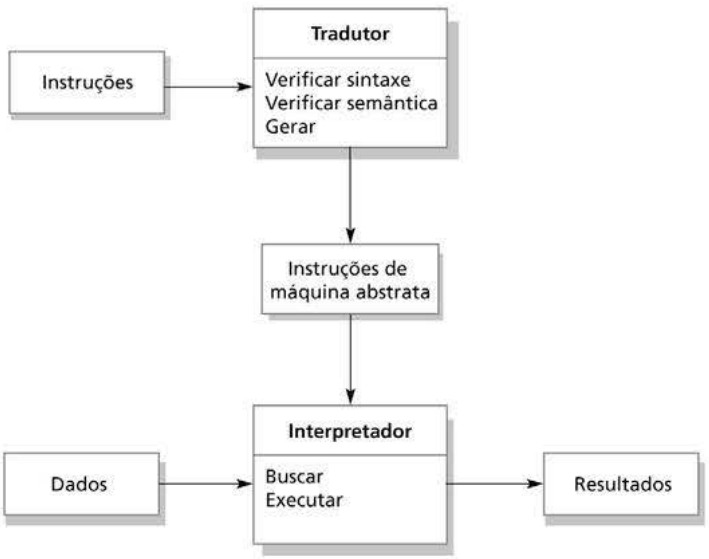
Os sistemas de processamento de linguagens aceitam uma linguagem natural ou artificial como entrada e geram alguma outra representação dessa linguagem como saída. Em engenharia de software, os sistemas de processamento de linguagens mais amplamente usados são os compiladores que traduzem uma linguagem artificial de programação de alto nível em código de máquina. Mas outros sistemas de processamento de linguagens traduzem uma descrição de dados XML em comandos para consultar um banco de dados e sistemas de processamento de linguagem natural que tentam traduzir uma linguagem em outra.

No nível mais abstrato, a arquitetura de um sistema de processamento de linguagens é ilustrada na Figura 13.11. As instruções descrevem o que precisa ser feito e são traduzidas em algum formato interno pelo tradutor. As instruções correspondem às instruções de máquina para uma máquina abstrata. Essas instruções são, então, interpretadas por outro componente que busca as instruções para execução e os executa usando, se necessário, dados do ambiente. A saída do processo é o resultado da interpretação das instruções sobre os dados de entrada. Naturalmente, para muitos compiladores, o interpretador é uma unidade de hardware que processa instruções de máquina e a máquina abstrata é um processador real. Contudo, para linguagens tal como Java, o interpretador é um componente de software.

Os sistemas de processamento de linguagens são usados em situações em que a maneira mais fácil de resolver um problema é especificar essa solução como um algoritmo ou como uma descrição de dados de sistema. Por exemplo, as ferramentas meta CASE são geradores de programas usados para criar ferramentas CASE específicas para apoiar os métodos de engenharia de software. As ferramentas meta CASE incluem uma descrição dos componentes de métodos, suas regras etc., escritos em uma linguagem com propósito específico que é classificada e analisada para configurar a ferramenta CASE gerada.

Figura 13.11

Arquitetura abstrata de um sistema de processamento de linguagens.



Os tradutores em um sistema de processamento de linguagens têm uma arquitetura genérica (Figura 13.12) que inclui os seguintes componentes:

1. Um analisador léxico, que obtém elementos da linguagem de entrada e os converte em um formato interno.
2. Uma tabela de símbolos, que mantém informações sobre os nomes de entidades (variáveis, nomes de classes, nomes de objetos etc.) usadas no texto que está sendo traduzido.
3. Um analisador sintático, que verifica a sintaxe da linguagem que está sendo traduzida. Ele usa uma gramática definida da linguagem e constrói uma árvore de sintaxe.
4. Uma árvore de sintaxe, que é uma estrutura interna que representa o programa que está sendo compilado.
5. Um analisador semântico, que usa informações da árvore de sintaxe e da tabela de símbolos para verificar a correção sintática do texto da linguagem de entrada.
6. Um gerador de código que 'caminha' pela árvore de sintaxe e gera o código de máquina abstrata.

Outros componentes que transformam a árvore de sintaxe podem ser incluídos para aumentar a eficiência e remover a redundância do código de máquina gerado. Em outros tipos de sistemas de processamento de linguagens, como um tradutor de linguagem natural, o código gerado é realmente o texto de entrada traduzido em uma outra linguagem.

Os componentes que constituem um sistema de processamento de linguagem podem ser organizados de acordo com modelos de arquitetura diferentes. Como Garlan e Shaw apontaram (Garlan e Shaw, 1993), os compiladores podem ser implementados com o uso de um modelo composto. Uma arquitetura de fluxo de dados pode ser usada com a tabela de símbolos que atua como um repositório para os dados compartilhados. As fases das análises léxica, sintática e semântica são organizadas seqüencialmente, conforme mostrado na Figura 13.12.

Esse modelo de fluxo de dados de compilação ainda é amplamente usado. É eficaz em ambientes em que os programas são compilados e executados sem a interação do usuário. É menos eficaz quando o compilador deve ser integrado a outras ferramentas de processamento de linguagens, como um sistema de edição estruturado, um debugger interativo ou uma impressora de programa. Os componentes genéricos do sistema podem, então, ser organizados em um modelo baseado em repositórios, conforme mostrado na Figura 13.13.

Essa figura mostra como um sistema de processamento de linguagens pode ser parte de um conjunto integrado de ferramentas de apoio à programação. Nesse exemplo, a tabela de símbolos e a árvore de sintaxe atuam como um repositório central de informações. As ferramentas e os fragmentos de ferramentas se comunicam através dele. Outras informações algumas vezes incorporadas a ferramentas, como a definição da gramática e a definição do formato de saída para o programa, foram retiradas das ferramentas e colocadas nos repositórios. Portanto, um editor dirigido à sintaxe pode verificar se a sintaxe de um programa está correta à medida que está sendo escrita, e uma impressora pode criar listagens de programa em um formato fácil de ser lido.

Figura 13.12

Modelo de fluxo de dados de um compilador.

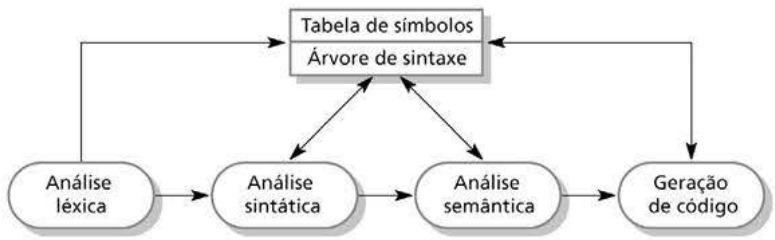


Figura 13.13

Modelo de repositório de um sistema de processamento de linguagens.

PONTOS-CHAVE

- Os modelos genéricos de arquiteturas de sistema de aplicações nos auxiliam a compreender a operação das aplicações, a comparar as aplicações de mesmo tipo, a validar projetos de sistema de aplicações e a avaliar os componentes de larga escala para reuso.
- Muitas aplicações pertencem a uma das quatro classes de aplicações genéricas ou são combinações dessas aplicações genéricas. Os quatro tipos de aplicações genéricas tratados neste capítulo são sistemas de processamento de dados, sistemas de processamento de transações, sistemas de processamento de eventos e sistemas de processamento de linguagens.
- Os sistemas de processamento de dados operam em lotes, e geralmente têm uma estrutura entrada-processo-saída. Os registros são inseridos no sistema, as informações são processadas e as saídas são geradas.
- Os sistemas de processamento de transações são sistemas interativos que permitem que informações no banco de dados sejam remotamente acessadas e modificadas por uma série de usuários. Os sistemas de informações e os sistemas de gerenciamento de recursos são exemplos de sistemas de processamento de transações.
- Os sistemas de processamento de eventos incluem sistemas de edição e sistemas de tempo real. Em um sistema de edição, os eventos de interface com o usuário são interpretados e uma estrutura de dados armazenada internamente é modificada. Os processadores de texto e sistemas de apresentação são exemplos de sistemas de edição.
- Os sistemas de processamento de linguagens são usados para traduzir textos de uma linguagem para outra e executar as instruções especificadas na linguagem de entrada. Eles incluem um tradutor e uma máquina abstrata que executa a linguagem gerada.

LEITURAS SUGERIDAS

O tópico de arquiteturas de aplicações foi muito negligenciado; os autores de livros e artigos sobre arquitetura de software tendem a enfocar os princípios abstratos ou arquiteturas de linhas de produto.

Databases and transaction processing: an application-oriented approach. Este não é realmente um livro sobre arquitetura de software, mas explica os princípios das aplicações de processamento de transações e as centradas em dados. (P. M. Lewis, et al., 2003, Addison-Wesley.)

Design and use of software architectures. Este livro adota uma abordagem de linha de produto para arquiteturas de software e, assim, explica a arquitetura da perspectiva da aplicação. (J. Bosch, 2000, Addison-Wesley.)

EXERCÍCIOS

13.1 Explique como as arquiteturas genéricas de aplicações descritas neste capítulo podem ser usadas para auxiliar o projetista a tomar decisões sobre o reuso de software.

13.2 Usando os quatro tipos básicos de aplicações apresentados neste capítulo, classifique os seguintes sistemas e explique sua classificação:

- Um sistema de ponto-de-venda em um supermercado.
- Um sistema que envia avisos sobre as assinaturas de revistas que estão com atraso dos pagamentos.
- Um sistema de álbum de fotografias que fornece alguns recursos para a restauração de fotografias antigas.
- Um sistema que lê páginas Web para usuários com deficiência visual.
- Um jogo interativo em que os personagens se movem, atravessam obstáculos e coletam tesouros.
- Um sistema de controle de estoques que mantém o acompanhamento de quais itens estão no estoque e gera pedidos automaticamente para novo estoque quando o nível fica abaixo de um certo valor.

13.3 Baseado em um modelo entrada-processo-saída, expanda a função **Calcular salário** da Figura 13.2 e desenhe um diagrama de fluxo de dados mostrando os cálculos realizados nessa função. Você precisa das seguintes informações para fazer isso:

- O registro de empregados identifica o grau de um empregado. Esse grau é usado para verificar a tabela de taxas de pagamento.
- Aos empregados abaixo de determinado grau podem ser pagas horas extras na mesma taxa que do pagamento normal por hora. As horas extras pagas são indicadas no registro do empregado.
- A quantidade de impostos deduzidos depende do código de impostos do empregado (indicado no registro) e seu salário anual. As deduções mensais para cada código e um salário-padrão estão indicados na tabela de impostos. Essas deduções variam para cima ou para baixo dependendo do relacionamento entre o salário real e o salário-padrão usado.

13.4 Explique por que o gerenciamento de transações é necessário em sistemas em que as entradas de usuário podem resultar em mudanças no banco de dados.

13.5 Usando o modelo básico de um sistema de informações conforme apresentado na Figura 13.6, mostre os componentes de um sistema de informações que permitem aos usuários visualizarem as informações sobre pousos e decolagens de vôos de um determinado aeroporto.

13.6 Usando a arquitetura de camadas mostrada na Figura 13.8, mostre os componentes de um sistema de gerenciamento de recursos que poderiam ser usados para cuidar de reservas de quartos de hotel.

13.7 Em um sistema de edição, todos os eventos da interface com o usuário podem ser traduzidos em comandos implícitos e explícitos. Explique por que, na Figura 13.10, o objeto **Evento** se comunica diretamente com a estrutura de dados de editor assim como o objeto **Comando**.

13.8 Modifique a Figura 13.10 e mostre a arquitetura genérica de um sistema de planilha eletrônica. Baseie seu projeto nas características de qualquer sistema de planilha que você tenha usado.

13.9 Qual é a função do componente árvore de sintaxe em um sistema de processamento de linguagens?

13.10 Usando o modelo genérico de um sistema de processamento de linguagens apresentado neste capítulo, projete a arquitetura de um sistema que aceita comandos de linguagem natural e traduz esses comandos em consultas de banco de dados em uma linguagem como SQL.

14

Projeto orientado a objetos

Objetivos

O objetivo deste capítulo é apresentar uma abordagem para projeto de software na qual o projeto é estruturado como objetos interativos. Após ler este capítulo, você:

- compreenderá como um projeto de software pode ser representado como um conjunto de objetos que interagem e que gerenciam seu próprio estado e operações;
- conhecerá as atividades mais importantes em um processo geral de projeto orientado a objetos;
- compreenderá os diferentes modelos que podem ser usados para documentar um projeto orientado a objetos;
- terá conhecido a representação desses modelos na linguagem de modelagem unificada (UML — Unified Modeling Language).

Conteúdo

14.1 Objetos e classes de objetos
14.2 Processo de projeto orientado a objetos
14.3 Evolução de projeto

Um sistema orientado a objetos é constituído de objetos que interagem e mantêm seu próprio estado local, fornecendo operações baseadas nesse estado (Figura 14.1). A representação do estado é privativa e não pode ser acessada diretamente de fora do objeto. Os processos de projeto orientados a objetos envolvem o projeto de classes de objeto e os relacionamentos entre essas classes. Essas classes definem os objetos do sistema e suas interações. Quando o projeto é realizado como um programa em execução, os objetos são criados dinamicamente a partir dessas definições de classes.

O projeto orientado a objetos é parte do desenvolvimento orientado a objetos no qual uma estratégia orientada a objetos é usada em todo o processo de desenvolvimento:

- *Análise orientada a objetos* concentra-se no desenvolvimento de um modelo orientado a objetos do domínio da aplicação. Os objetos nesse modelo refletem as entidades e as operações associadas ao problema a ser resolvido.
- *Projeto orientado a objetos* concentra-se no desenvolvimento de um modelo orientado a objetos de um sistema de software para implementar os requisitos identificados. Os objetos em um projeto orientado a objetos estão relacionados à solução do problema. Pode haver

relacionamentos próximos entre alguns objetos do problema e alguns objetos da solução, mas o projetista precisa, inevitavelmente, adicionar novos objetos e transformar os objetos do problema para implementar a solução.

- *Programação orientada a objetos* concentra-se em realizar um projeto de software usando uma linguagem de programação orientada a objetos, tal como Java. Uma linguagem de programação orientada a objetos fornece construções que definem classes de objeto e um sistema de *run-time* para criar objetos a partir dessas classes.

A transição entre esses estágios de desenvolvimento deve, idealmente, ser contínua, com notações compatíveis usadas em cada estágio. Passar ao próximo estágio envolve o refinamento do estágio anterior pela adição de detalhes nas classes de objetos existentes e a criação de novas classes para fornecer funcionalidade adicional. Como as informações estão ocultas dentro dos objetos, decisões detalhadas de projeto sobre a representação de dados podem ser adiadas até que o sistema seja implementado. Em alguns casos, decisões sobre a distribuição de objetos e se os objetos devem ser seqüenciais ou concorrentes também podem ser adiadas.

Isso significa que os projetistas de software podem criar projetos que podem ser adaptados a ambientes de execução diferentes. Isso é exemplificado pela abordagem de arquitetura dirigida a modelos (MDA — *Model Driven Architecture*), que propõe que os sistemas devem ser projetados explicitamente em dois níveis (Kleppe et al., 2003), um nível independente de implementação e um nível dependente de implementação. Um modelo abstrato do sistema é projetado no nível independente de implementação e isso é mapeado para um modelo dependente de plataforma com mais detalhes que pode ser usado como base para a geração de código. No momento em que este livro estava sendo escrito, a abordagem MDA ainda era experimental e não estava claro como ela poderia ser amplamente adotada.

Os sistemas orientados a objetos são mais fáceis de alterar do que os sistemas desenvolvidos usando outras abordagens, pois os objetos são independentes. Eles podem ser compreendidos e modificados como entidades independentes. A mudança de implementação de um objeto ou a adição de serviços não deve afetar outros objetos de sistema. Devido aos objetos serem associados a coisas, há muitas vezes um mapeamento claro entre entidades do mundo real (como componentes de hardware) e seus objetos de controle no sistema. Isso aprimora a capacidade de compreensão e, portanto, a facilidade de manutenção do projeto.

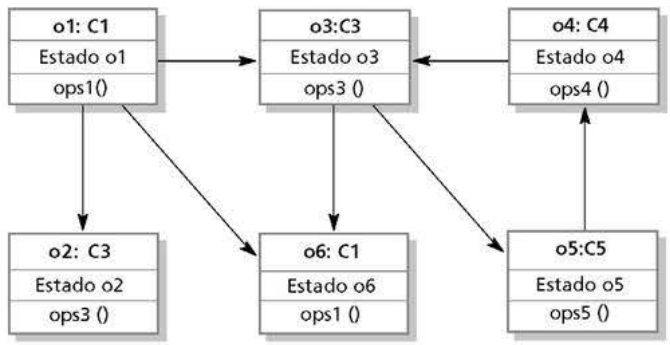
Objetos são, potencialmente, componentes reusáveis porque eles são encapsulamentos independentes de estado e operações. Os projetos podem ser desenvolvidos usando objetos criados em projetos anteriores. Isso reduz os custos de projeto, programação e validação. Isso pode também conduzir ao uso de objetos-padrão (portanto, aprimorando a capacidade de compreensão do projeto) e redução dos riscos envolvidos no desenvolvimento de software. Contudo, conforme explicado nos capítulos 18 e 19, o reuso é algumas vezes mais implementado com a utilização de conjuntos de objetos (componentes ou *frameworks*) em vez de objetos individuais.

Vários métodos de projeto orientado a objetos foram propostos (Coad e Yourdon, 1990; Robinson, 1992; Jacobson, et al., 1993; Graham, 1994; Booch, 1994). A UML é uma unificação de notações usadas nesses métodos. O Rational Unified Process (RUP), que expliquei no Capítulo 4, foi projetado para explorar os modelos que podem ser expressos em UML (Rumbaugh, et al., 1999). A UML é usada em todo o capítulo.

Conforme explicado no Capítulo 17, o desenvolvimento de sistema baseado em um extenso projeto antecipado pode receber críticas, pois o extenso esforço de análise e projeto não é adequado ao desenvolvimento e entregas incrementais. Os chamados métodos ágeis foram desenvolvidos para resolver esse problema, e esses métodos reduzem drasticamente ou eliminam completamente a atividade de projeto orientado a objetos. Minha visão sobre isso é que projeto extenso e 'pesado' não é necessário para sistemas de empresas de pequeno e médio portes. No entanto, para sistemas de grande porte, particularmente sistemas críticos, é essencial garantir que as equipes que trabalham em partes diferentes do sistema estejam coordenadas. Por esse motivo, não usei os exemplos anteriores de sistema de biblioteca ou de bomba de insulina neste capítulo porque estes são sistemas relativamente pequenos. Em vez disso, uso um exemplo que é parte de um sistema muito maior no qual o projeto inicial orientado a objetos é mais útil.

Figura 14.1

Sistema constituído de objetos interativos.



Essa visão reflete-se, em alguns aspectos, no Rational Unified Process, que é direcionado ao desenvolvimento iterativo e a entregas incrementais de sistemas de software de grande porte. Esse é um processo de desenvolvimento iterativo baseado em casos de uso para expressar requisitos e projeto orientado a objetos, com um foco específico no projeto centrado em arquitetura.

O processo de projeto explicado na Seção 14.2 tem algumas coisas em comum com o RUP, mas com menos ênfase no desenvolvimento dirigido a casos de uso. Os casos de uso significam, com certeza, que o projeto está centrado no usuário e baseia-se nas interações do usuário com o sistema. Contudo, a representação de requisitos dos stakeholders que não sejam usuários diretos do sistema como casos de uso é mais difícil. Casos de uso, certamente, têm um papel importante na análise de projeto orientado a objetos, mas precisam ser suplementados com outras técnicas para descoberta de requisitos de sistema indiretos e não funcionais.

14.1 Objetos e classes de objetos

Os termos *objeto* e *orientado a objetos* são aplicados a tipos diferentes de entidades, métodos de projeto, sistemas e linguagens de programação. Há um consenso geral de que um objeto é um encapsulamento de informações, e isso se reflete em minha definição de objeto e classe de objeto:

Um objeto é uma entidade que possui um estado e um conjunto definido de operações definidas para funcionar nesse estado. O estado é representado como um conjunto de atributos de objeto. As operações associadas ao objeto fornecem serviços a outros objetos (clientes) que solicitam esses serviços quando alguma computação é necessária.

Os objetos são criados de acordo com uma definição de classe de objeto. Uma definição de classe de objeto funciona tanto como uma especificação quanto como um template para criação de objetos. Essa definição inclui declarações de todos os atributos e operações que devem ser associados a um objeto dessa classe.

Na UML, uma classe de objeto é representada como um retângulo identificado com duas seções. Os atributos do objeto são relacionados na seção superior. As operações associadas ao objeto são definidas na seção inferior. A Figura 14.2 ilustra essa notação usando uma classe de objeto que modela um funcionário em uma organização. A UML usa o termo *operação* para definir a especificação de uma ação e o termo *método* para se referir à implementação de uma operação.

A classe **Funcionário** define alguns atributos que mantêm informações sobre os funcionários incluindo seu nome e endereço, número da previdência social, código de contribuinte etc. As reticências (...) indicam que existem mais atributos associados à classe do que os que foram mostrados. As operações associadas ao objeto são **join** (chamada quando um funcionário entra na organização), **leave** (chamada quando um funcionário deixa a organização), **retire** (chamada quando o funcionário se aposenta pela organização) e **changeDetails** (chamada quando alguma informação do funcionário precisa ser modificada).

Os objetos se comunicam por meio da solicitação de serviços (chamadas de métodos) de outros objetos e, se necessário, por meio da troca de informações necessárias para o fornecimento do serviço. As cópias das informações necessárias para executar o serviço e os resultados da execução do serviço são passados como parâmetros. Veja, na página seguinte, alguns exemplos desse estilo de comunicação.

Figura 14.2

Objeto Funcionário.

```
// Chamar um método associado a um objeto buffer que retorna o próximo valor
// no buffer
v = circularBuffer.Get () ;
// Chamar o método associado a um objeto termostato que ajuste a
// temperatura a ser mantida
thermostat.setTemp (20) ;
```

Em sistemas baseados em serviços, as comunicações de objeto são implementadas diretamente como mensagens de texto XML que os objetos trocam entre si. O objeto receptor analisa a mensagem, identifica o serviço e os dados associados e executa o serviço solicitado. Contudo, quando os objetos coexistem no mesmo programa, as chamadas de métodos são implementadas como chamadas de procedimento ou de função em uma linguagem como C.

Quando as solicitações de serviço são implementadas dessa maneira, a comunicação entre objetos é síncrona, ou seja, o objeto chamador espera pela conclusão da solicitação de serviço. No entanto, se os objetos forem implementados como processos concorrentes ou *threads*, a comunicação do objeto poderá ser assíncrona. O objeto chamador pode continuar em operação enquanto o serviço solicitado está em execução. Explico como os objetos podem ser implementados como processos concorrentes mais adiante nesta seção.

Conforme discuti no Capítulo 8, no qual descrevi uma série de possíveis modelos de objeto, as classes de objetos podem ser organizadas em uma estrutura de generalização ou de herança que mostra o relacionamento entre classes de objetos gerais ou mais específicas. A classe de objeto mais específica é completamente consistente com a classe-pai, mas inclui outras informações. Na UML, a seta que aponta de uma entidade de classe para sua classe-pai indica a generalização. Em linguagens de programação orientadas a objetos, a generalização é implementada por meio de herança. A classe-filho herda os atributos e as operações da classe-pai.

A Figura 14.3 mostra um exemplo de uma hierarquia de classes de objeto em que são mostradas classes diferentes de funcionário. As classes mais inferiores da hierarquia têm os mesmos atributos e operações que as classes-pai, mas podem acrescentar novos atributos e operações ou modificar alguns deles com base nas classes-pai. Isso significa que existe um caminho único de intercâmbio. Se o nome de uma classe-pai for usado em um modelo, o objeto no sistema pode ser definido como dessa classe ou de qualquer uma das classes-filho.

A classe **Gerente**, na Figura 14.3, tem todos os atributos e operações da classe **Funcionário**, mas possui, adicionalmente, dois novos atributos que registram os orçamentos controlados pelo gerente e a data na qual o gerente foi designado para uma função específica de gerenciamento. Da mesma maneira, a classe **Programador** acrescenta novos atributos que definem o projeto no qual o programador está trabalhando e as habilidades em linguagem de programação que ele possui. Objetos da classe **Gerente** ou **Programador** podem, portanto, ser usados em qualquer lugar onde um objeto da classe **Funcionário** for necessário.

Os objetos que sejam membros de uma classe de objeto participam de relacionamentos com outros objetos. Esses relacionamentos podem ser modelados pela descrição de associações entre as classes de objeto. Na UML, as associações são indicadas por uma linha entre as classes de objeto que podem, opcionalmente, conter anotações com informações sobre a associação. Isso é ilustrado na Figura 14.4, que mostra a associação entre os objetos da classe **Funcionário** e os objetos da classe **Departamento**, e entre os objetos da classe **Funcionário** e os objetos da classe **Gerente**.

Figura 14.3

Hierarquia de generalização.

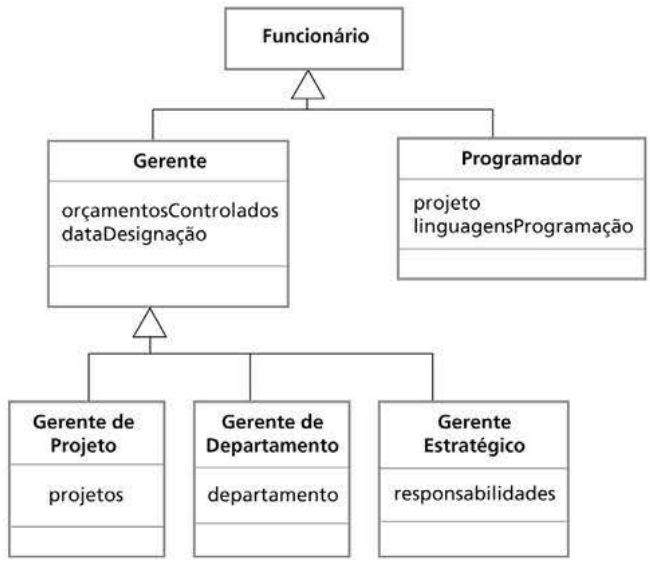
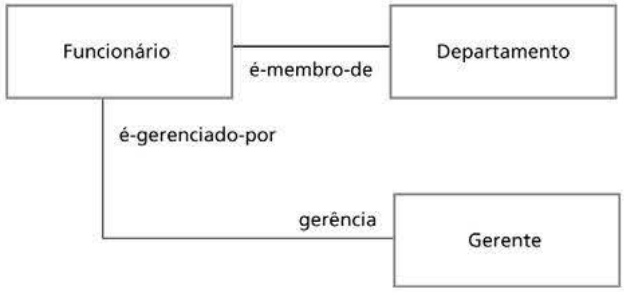


Figura 14.4
Modelo de associação.



A associação é um relacionamento muito genérico e, freqüentemente, é usada na UML para indicar que um atributo de um objeto é um objeto associado ou que a implementação de um método de objeto conta com o objeto associado. Contudo, em princípio, pelo menos, qualquer tipo de associação é possível. Uma das associações mais comuns é a agregação, que mostra como objetos podem ser compostos de outros objetos. Veja o Capítulo 8 para saber mais sobre esse tipo de associação.

14.1.1 Objetos concorrentes

Conceitualmente, um objeto solicita um serviço de um outro objeto enviando uma mensagem de 'solicitação de serviço' para esse objeto. Não existe requisito para execução em série, na qual um objeto aguarda o término de um serviço solicitado. Conseqüentemente, o modelo geral de interação de objetos permite que os objetos sejam executados concorrentemente como processos paralelos. Esses objetos podem ser executados no mesmo computador, ou como objetos distribuídos, em máquinas diferentes.

Na prática, a maioria das linguagens de programação orientadas a objeto tem como default um modelo de execução serial no qual solicitações de serviços de objeto são implementadas do mesmo modo que as chamadas de funções. Portanto, quando um objeto chamado **theList** é criado a partir de uma classe de objeto normal, escrevemos em Java:

theList.append (17)

Isso chama o método **append** associado ao objeto **theList** para acrescentar o elemento 17 em **theList**, e a execução do objeto chamador é suspensa até que a operação append seja completada. No entanto, o Java inclui um mecanismo muito simples (*threads*) que permite criar objetos executados concorrentemente. Os threads são criados em Java usando a classe **Thread** built-in como uma classe-pai em uma declaração de classe. Os threads devem incluir o método chamado **run**, iniciado pelo sistema de run-time do Java, quando os objetos definidos como threads são criados. Portanto, é fácil obter um projeto orientado a objetos e criar uma implementação na qual os objetos sejam processos concorrentes.

Existem dois tipos de implementação de objetos concorrentes:

1. *Servidores* na qual o objeto é implementado como um processo paralelo com métodos que correspondem às operações de objeto definidas. Os métodos são iniciados em resposta a uma mensagem externa e podem ser executados paralelamente com métodos associados a outros objetos. Quando eles concluírem sua operação, o objeto suspenderá a execução e aguardará outras solicitações por serviço.
2. *Objetos ativos* na qual o estado do objeto pode ser modificado pela execução de operações internas dentro do próprio objeto. O processo que representa o objeto executa continuamente essas operações e, assim, nunca suspende sua própria execução.

Os servidores são muito úteis em um ambiente distribuído, no qual os objetos chamador e chamado podem ser executados em computadores diferentes. O tempo de resposta para o serviço solicitado é imprevisível e, assim, sempre que possível, você deve projetar o sistema de modo que o objeto que solicitou um serviço não precise aguardar a conclusão desse serviço. Eles podem também ser usados em uma única máquina, na qual um serviço toma algum tempo para ser concluído (por exemplo, a impressão de um documento) e muitos objetos podem solicitar o serviço.

Objetos ativos são usados quando um objeto precisa atualizar seu próprio estado em intervalos definidos. Isso é comum em sistemas de tempo real, no qual os objetos são associados a dispositivos de hardware que coletam informações sobre o ambiente do sistema. Os métodos do objeto permitem que outros objetos acessem as informações de estado.

O Quadro 14.1 mostra como um objeto ativo pode ser definido e implementado em Java. A classe de objeto representa um *transponder* de uma aeronave. O transponder mantém o controle da posição da aeronave usando um sistema de navegação por satélite. Ele pode responder a mensagens dos computadores de controle de tráfego aéreo. O aparelho

Quadro 14.1

Implementação de um objeto ativo usando threads em Java.

```
class Transponder extends Thread {

    Position currentPosition ;
    Coords c1, c2 ;
    Satellite sat1, sat2 ;
    Navigator theNavigator ;

    public Position givePosition ()
    {
        return currentPosition ;
    }

    public void run ()
    {
        while (true)
        {
            c1 = sat1.position () ;
            c2 = sat2.position () ;
            currentPosition = theNavigator.compute (c1, c2) ;
        }
    }
} //Transponder
```

fornece a posição real da aeronave em resposta a uma solicitação do método **givePosition**. Esse objeto é implementado como um thread no qual um loop contínuo no método **run** inclui código para calcular a posição da aeronave usando sinais dos satélites.

14.2 Processo de projeto orientado a objetos

Nesta seção, ilustro o processo de projeto orientado a objetos desenvolvendo um exemplo de projeto para o software de controle embutido em uma estação meteorológica automatizada. Conforme expliquei na introdução, existem vários métodos de projeto orientado a objetos sem que haja um método ou processo de projeto que seja definitivamente 'o melhor'. O processo abordado nesta seção é um processo que incorpora atividades comuns à maioria dos processos de projeto orientados a objetos (OOD — Object-Oriented Design).

O processo geral que uso nesta seção para projeto orientado a objetos tem uma série de estágios:

1. Compreender e definir o contexto e os modos de uso do sistema.
2. Projetar a arquitetura do sistema.
3. Identificar os objetos principais do sistema.
4. Desenvolver modelos de projeto.
5. Especificar interfaces de objetos.

Eu não ilustrei isso deliberadamente como um diagrama de processo simples, pois isso implicaria uma nítida seqüência de atividades nesse processo. De fato, todas as atividades citadas são intercaladas e, assim, se influenciam reciprocamente. Os objetos são identificados e as interfaces são completa ou parcialmente especificadas quando a arquitetura do sistema é definida. À medida que os modelos de objeto são produzidos, essas definições individuais de objeto podem se refinadas, o que conduz a mudanças na arquitetura do sistema.

Esses estágios são explicados separadamente no processo de projeto mais adiante nesta seção. No entanto, você não deve supor a partir disso que projetar é um processo simples e bem estruturado. Na realidade, você desenvolve um projeto que propõe soluções e refina essas soluções à medida que as informações se tornam disponíveis. Você precisa, inevitavelmente, retroceder e tentar novamente quando surgem problemas. Algumas vezes, é necessário explorar as opções detalhadamente para ver se elas funcionam e, em outras, é preciso ignorar os detalhes até o fim do processo.

Ilustro essas atividades de processo desenvolvendo um exemplo de projeto orientado a objetos. Esse exemplo é parte de um sistema para criação de mapas meteorológicos que usa dados meteorológicos coletados automaticamente. Os requisitos

detalhados para tal sistema de mapeamento meteorológico ocupariam muitas páginas. Contudo, uma arquitetura geral do sistema pode ser desenvolvida com uma descrição relativamente breve do sistema:

Um sistema de mapeamento meteorológico é necessário para gerar mapas meteorológicos regularmente usando dados coletados de estações remotas, sem funcionários, e de outras fontes de dados, como observatórios, balões e satélites. As estações meteorológicas transmitem seus dados para um computador local em resposta à solicitação dessa máquina.

O sistema de computador local valida os dados coletados e integra os dados de diferentes fontes. Os dados integrados são arquivados e, usando dados desse arquivo e um banco de dados de mapas digitalizados, um conjunto de mapas meteorológicos locais é criado. Os mapas podem ser impressos para distribuição em uma impressora de mapas especial e ser apresentados em diversos formatos.

Essa descrição mostra que parte do sistema geral se dedica à coleta de dados, parte à integração de dados de fontes diferentes, parte ao arquivamento desses dados e parte à criação de mapas meteorológicos. A Figura 14.5 ilustra uma possível arquitetura de sistema que pode ser derivada dessa descrição. É uma arquitetura em camadas (explicadas no Capítulo 11) que reflete os estágios de processamento no sistema, denominados coleta de dados, integração de dados, arquivamento de dados e geração de mapas. Uma arquitetura em camadas é adequada neste caso, pois cada estágio conta com o processamento do estágio anterior para sua operação.

Na Figura 14.5, mostrei as camadas e incluí o nome da camada em um símbolo de *package* UML, indicado como um subsistema. Um package UML representa um conjunto de objetos e outros packages. Usei isso nesta seção para mostrar que cada camada inclui uma série de outros componentes.

Na Figura 14.6, expandi esse modelo abstrato de arquitetura mostrando os componentes dos subsistemas. Esses subsistemas são ainda abstratos e foram derivados das informações contidas na descrição do sistema. Continuo com o exemplo de projeto que enfoca o subsistema da estação meteorológica que é parte da camada de coleta de dados.

Figura 14.5

Arquitetura em camadas de um sistema de mapeamento meteorológico.

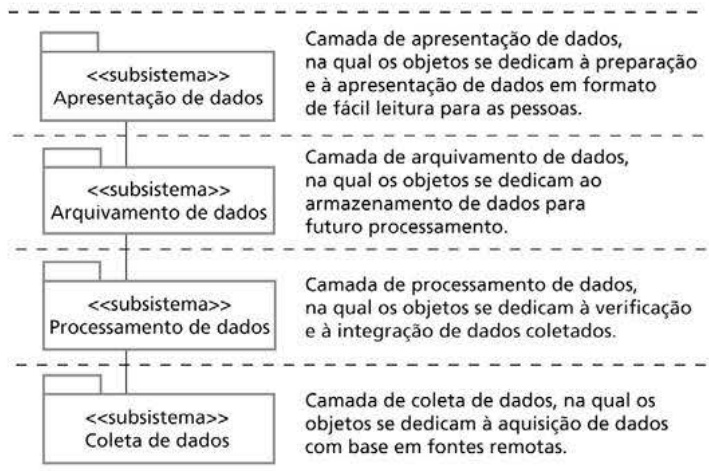
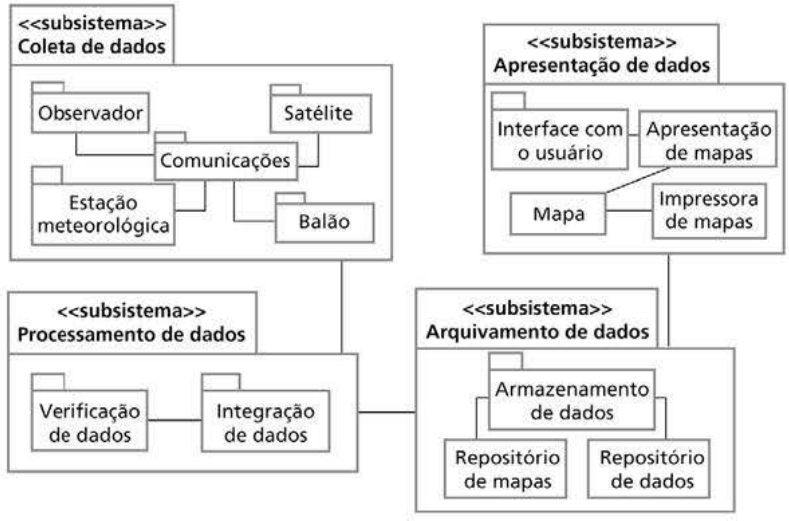


Figura 14.6

Subsistemas no sistema de mapeamento meteorológico.



14.2.1 Contexto de sistema e modelos de uso

O primeiro estágio em qualquer processo de projeto de software consiste em desenvolver uma compreensão dos relacionamentos entre o software que está sendo projetado e seu ambiente externo. Essa compreensão é necessária para auxiliar em sua decisão de como prover a funcionalidade de sistema necessária e como estruturar o sistema para comunicação com seu ambiente.

O contexto e o modelo de uso de sistema representam dois modelos complementares de relacionamentos entre um sistema e seu ambiente:

1. O contexto de sistema é um modelo estático que descreve os outros sistemas nesse ambiente.
2. O modelo de uso de sistema é um modelo dinâmico que descreve como o sistema realmente interage com seu ambiente.

O modelo de contexto de um sistema pode ser representado por meio de associações (veja a Figura 14.4) nas quais um simples diagrama de blocos da arquitetura geral do sistema é produzido. Essa arquitetura, então, é desenvolvida por meio da derivação de um modelo de subsistema que usa packages UML, conforme mostrado na Figura 14.6. Esse modelo mostra que o contexto do sistema da estação meteorológica está dentro de um subsistema dedicado à coleta de dados. Também são mostrados outros subsistemas que constituem o sistema de mapeamento meteorológico.

Quando você modela as interações de um sistema com seu ambiente, é necessário usar uma abordagem que não inclua muitos detalhes. A abordagem proposta no RUP é desenvolver um modelo de caso de uso, com cada caso de uso representando uma interação com o sistema. Nos modelos de casos de uso (também explicados no Capítulo 7), cada interação possível é representada por uma elipse e a entidade externa envolvida na interação é representada pela figura de um boneco. No caso do sistema da estação meteorológica, essa entidade externa não é uma pessoa, mas sim o sistema de processamento de dados meteorológicos.

O modelo de caso de uso para a estação meteorológica é mostrado na Figura 14.7. Esse modelo mostra que a estação meteorológica interage com entidades externas para a inicialização e desativação, para exclusão dos dados meteorológicos coletados e para teste e ajuste dos instrumentos.

Cada um desses casos de uso pode ser descrito em uma linguagem natural estruturada. Isso auxilia os projetistas a identificar objetos do sistema e permite uma compreensão do que o sistema destina-se a fazer. Uso um formulário padrão para essa descrição que identifica claramente quais informações são trocadas, como a interação é iniciada etc. Isso é mostrado na Tabela 14.1, que descreve o caso de uso **Relatar** da Figura 14.7.

A descrição de caso de uso ajuda a identificar objetos e operações do sistema. A partir da descrição do caso de uso **Relatar**, é óbvio que os objetos que representam os instrumentos que coletam dados meteorológicos serão necessários, bem como um objeto que representa o resumo de dados meteorológicos. São necessárias, também, as operações para solicitar dados meteorológicos e para enviar esses dados.

14.2.2 Projeto de arquitetura

Após a definição das interações entre o sistema de software que está sendo projetado e o ambiente do sistema, você pode usar essas informações como base para o projeto de arquitetura do sistema. Naturalmente, é necessário combinar essas informações com seu conhecimento geral de princípios de projeto de arquitetura e com conhecimento de domínio mais detalhados.

Figura 14.7

Casos de uso para a estação meteorológica.

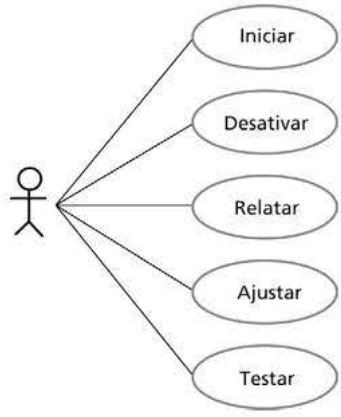


Tabela 14.1 Descrição do caso de uso **Relatar**

Sistema	Estação meteorológica
Caso de uso	Relatar
Agentes	Sistema de coleta de dados meteorológicos, Estação meteorológica
Dados	A estação meteorológica envia um resumo de dados meteorológicos, coletados a partir dos instrumentos, ao sistema de coleta de dados meteorológicos. Os dados enviados são as temperaturas máxima, mínima e média do solo e do ar, pressões máxima, mínima e média do ar, velocidades máxima, mínima e média do vento, precipitação total de chuva e direção do vento conforme amostragem em intervalos de cinco minutos.
Estímulo	O sistema de coleta de dados meteorológicos estabelece um link via modem com a estação meteorológica e solicita a transmissão dos dados.
Resposta	Os dados resumidos são enviados ao sistema de coleta de dados meteorológicos.
Comentários	As estações meteorológicas geralmente devem relatar uma vez a cada hora, mas essa freqüência pode diferir de uma estação para outra e pode ser modificada no futuro.

A estação meteorológica automatizada é um sistema relativamente simples, e sua arquitetura pode novamente ser representada como um modelo de camadas. Isso é mostrado na Figura 14.8 como três packages UML dentro de um package mais geral denominado *estação meteorológica*. Observe como usei as notações UML (texto em caixas com um canto dobrado) para fornecer informações adicionais.

As três camadas no software da estação meteorológica são:

1. A *camada interface* que cuida de todas as comunicações com outras partes do sistema e do fornecimento de interfaces externas do sistema.
2. A *camada coleta de dados* cuida do gerenciamento da coleta de dados dos instrumentos e do resumo dos dados meteorológicos antes da transmissão para o sistema de mapeamento.
3. A *camada de instrumentos* que é um encapsulamento de todos os instrumentos usados para coleta de dados brutos sobre as condições climáticas.

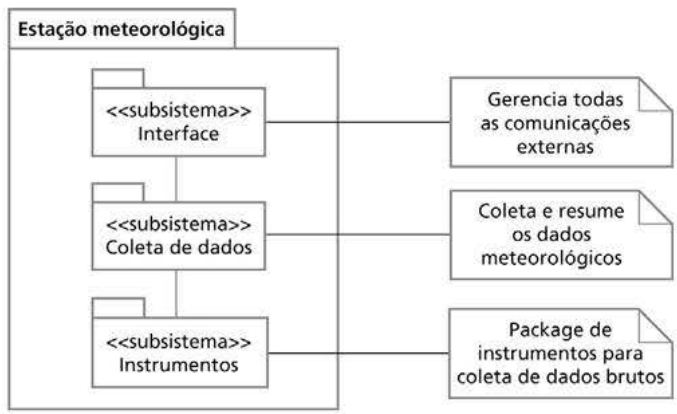
Em geral, você deve tentar decompor um sistema de modo que arquiteturas sejam o mais simples possível. Uma boa regra prática é que não deve haver mais do que sete entidades fundamentais em um modelo de arquitetura. Cada uma dessas entidades pode ser descrita separadamente mas, naturalmente, você pode revelar a estrutura das entidades como fiz na Figura 14.6.

14.2.3 Identificação de objetos

Neste estágio do processo de projeto, você deve ter algumas idéias sobre os objetos essenciais do sistema que está projetando. No sistema da estação meteorológica, é claro que os instrumentos devem ser objetos, e você precisa de pelo menos um objeto para cada um dos níveis da arquitetura. Isso reflete um princípio geral de que os objetos tendem

Figura 14.8

Arquitetura da estação meteorológica.



a emergir durante o processo de projeto. Contudo, você também precisa, geralmente, procurar e documentar outros objetos que podem ser relevantes.

Embora eu tenha denominado esta seção como 'identificação de objetos', na prática, esse processo se dedica à identificação de classes de objetos. O projeto é descrito em termos dessas classes. Inevitavelmente, você precisa refinar as classes de objeto identificadas inicialmente e retornar a esse estágio do processo à medida que desenvolver uma compreensão mais profunda do projeto.

Várias propostas têm sido feitas sobre como identificar classes de objeto:

1. Usar uma análise gramatical de uma descrição em linguagem natural de um sistema. Os objetos e atributos são nomes, as operações ou serviços são verbos (Abbott, 1983). Essa abordagem foi incorporada ao método HOOD para projetos orientados a objetos (Robinson, 1992) amplamente usado na indústria aeroespacial européia.
2. Usar entidades tangíveis (coisas) no domínio de aplicação como aeronave, funções como gerente, eventos como solicitação, interações como reuniões, locais como escritórios, unidades organizacionais como empresas etc. (Shlaer e Mellor, 1988; Coad e Yourdon, 1990; Wirfs-Brock, et al., 1990). Sustentar isso identificando estruturas de repositório (estruturas de dados abstratos) no domínio da solução que poderiam ser necessárias para apoiar esses objetos.
3. Usar uma abordagem comportamental, na qual o projetista compreende primeiro o comportamento geral do sistema. Os diversos comportamentos são atribuídos a diferentes partes do sistema e uma compreensão é derivada de quem inicia e participa desses comportamentos. Os participantes que desempenham papéis significativos são reconhecidos como objetos (Rubin e Goldberg, 1992).
4. Usar análise baseada em cenários, na qual diversos cenários de uso do sistema são identificados e analisados um de cada vez. Quando cada cenário é analisado, a equipe responsável pela análise deve identificar os objetos, atributos e operações necessários. Um método de análise chamado cartões CRC, no qual analistas e projetistas assumem o papel de objetos, é eficaz no apoio a essa abordagem baseada em cenários (Beck e Cunningham, 1989).

Essas abordagens nos ajudam a iniciar a identificação de objetos. Na prática, pode ser necessário usar diversas fontes de conhecimento para descobrir classes de objeto. As classes de objeto, atributos e operações inicialmente identificados baseados em uma descrição informal do sistema podem ser o ponto de partida para o projeto. Informações adicionais, obtidas com o conhecimento do domínio da aplicação ou análise de cenários, podem ser usadas para refinar e estender os objetos iniciais. Essas informações podem ser coletadas a partir de documentos de requisitos, de discussões com os usuários e de uma análise de sistemas existentes.

Usei uma abordagem híbrida nesta seção para identificar os objetos da estação meteorológica. Não há espaço para descrever todos os objetos, mas são apresentadas cinco classes de objeto na Figura 14.9. **Termômetro de solo**, **Anemômetro** e **Barômetro** representam os objetos do domínio de aplicação, e os objetos **EstaçãoMeteorológica** e **DadosMeteorológicos** foram identificados pelas descrições de sistema e de cenários (casos de uso).

Esses objetos estão relacionados aos níveis da arquitetura de sistema.

Figura 14.9

Exemplos de classes de objeto do sistema da estação meteorológica.

1. A classe de objeto **EstaçãoMeteorológica** fornece uma interface básica da estação meteorológica com seu ambiente. Suas operações, portanto, refletem as interações mostradas na Figura 14.7. Neste caso, usei uma única classe de objeto para englobar todas essas interações, mas em outros projetos podem ser usadas várias classes diferentes para a interface do sistema.
2. A classe de objeto **DadosMeteorológicos** engloba os dados resumidos obtidos com base em instrumentos da estação meteorológica. Suas operações associadas são dedicadas à coleta e ao resumo dos dados necessários.
3. As classes de objeto **Termômetro de solo**, **Anemômetro** e **Barômetro** estão diretamente relacionadas aos instrumentos do sistema. Elas refletem as entidades de hardware tangíveis do sistema e as operações dedicam-se ao controle desse hardware.

Neste estágio do processo de projeto, o conhecimento do domínio de aplicação pode ser usado para identificar objetos e serviços adicionais. Sabemos que as estações meteorológicas estão freqüentemente localizadas em lugares remotos e incluem vários instrumentos que algumas vezes não funcionam. As falhas de instrumentos devem ser relatadas automaticamente. Isso significa que são necessários atributos e operações para verificar o correto funcionamento dos instrumentos. Obviamente, existem muitas estações meteorológicas remotas. Você precisa identificar os dados coletados de cada estação, sendo assim cada uma delas deve ter seu próprio identificador.

Neste exemplo, decidi que os objetos associados a cada instrumento não devem ser objetos ativos. A operação coletar na classe de objeto **DadosMeteorológicos** chama objetos de instrumentos para fazer leituras quando necessário. Os objetos ativos incluem seu próprio controle e, nesse caso, significa que cada instrumento decide quando fazer as leituras. A desvantagem disso é que, se alguma decisão for tomada para mudar a freqüência de coleta de dados ou se diferentes estações meteorológicas coletarem dados de modo diferente, novas classes de objeto precisarão ser introduzidas. Ao fazer com que os objetos de instrumentos realizem leituras quando necessário, quaisquer mudanças na estratégia de coleta podem ser facilmente implementadas sem modificar os objetos associados aos instrumentos.

14.2.4 Modelos de projeto

Modelos de projeto mostram os objetos ou as classes de objeto em um sistema e, quando apropriado, os relacionamentos entre essas entidades. Os modelos de projeto são essencialmente o próprio projeto. Eles são a ponte entre os requisitos do sistema e a implementação do sistema. Isso significa que existem requisitos conflitantes nesses modelos. Eles precisam ser abstratos de modo que os detalhes desnecessários não ocultem os relacionamentos entre eles e os requisitos de sistema. No entanto, eles também precisam incluir detalhes suficientes para que os programadores possam tomar decisões sobre a implementação.

Em geral, esse conflito é contornado pelo desenvolvimento de modelos com diferentes níveis de detalhes. Quando houver vínculos diretos entre engenheiros de requisitos, projetistas e programadores, os modelos abstratos podem ser tudo de que se precisa. As decisões específicas de projeto podem ser tomadas à medida que o sistema é implementado. Quando os vínculos entre os especificadores de sistema, projetistas e programadores forem indiretos (por exemplo, um sistema está sendo projetado em uma parte da organização, mas implementado em qualquer outro lugar), modelos mais detalhados serão necessários.

Um passo importante no processo de projeto, portanto, é decidir quais modelos de projeto são necessários e o nível de detalhamento desses modelos. Isso depende do tipo de sistema que está sendo desenvolvido. Um sistema de processamento de dados seqüencial será projetado de maneira diferente de um sistema de tempo real incorporado, portanto modelos de projeto diferentes serão usados. Existem muito poucos sistemas em que todos os modelos são necessários. A minimização do número de modelos produzidos reduz os custos do projeto e o tempo necessário para concluir o processo de projeto.

Existem dois tipos de modelos de projeto que devem ser normalmente produzidos para descrever um projeto orientado a objetos:

1. *Modelos estáticos* descrevem a estrutura estática do sistema usando classes de objeto e seus relacionamentos. Os relacionamentos importantes que podem ser documentados nesse estágio são os de generalização, do tipo utiliza/é-utilizado-por e de composição.
2. *Modelos dinâmicos* descrevem a estrutura dinâmica do sistema e mostram as interações entre os objetos de sistema (não as classes de objeto). As interações que podem ser documentadas incluem a seqüência de solicitações de serviço feitas pelos objetos e o modo como o estado do sistema está relacionado a essas interações de objeto.

A UML fornece doze modelos estáticos e dinâmicos diferentes que podem ser produzidos para documentar um projeto. Não há espaço suficiente para tratar de todos esses modelos, nem todos são adequados ao exemplo da estação meteorológica. Os modelos que uso nesta seção são:

1. *Modelos de subsistemas* que mostram os agrupamentos lógicos de objetos em subsistemas coerentes. Esses modelos são representados por meio de um formato de diagrama de classes no qual cada subsistema é mostrado como um pacote. Os modelos de subsistema são modelos estáticos.
2. *Modelos de seqüência* que mostram a seqüência das interações entre objetos. Esses modelos são representados por meio de um diagrama de seqüência ou colaboração da UML. Os modelos de seqüência são modelos dinâmicos.
3. *Modelos de máquina de estado* que mostram como objetos individuais mudam seu estado em resposta aos eventos. Esses modelos são representados por meio de diagramas *statechart* da UML. Os modelos de máquina de estado são modelos dinâmicos.

Já expliquei outros modelos que podem ser desenvolvidos para o projeto e análise orientados a objetos. Os modelos de caso de uso mostram as interações com o sistema (Figura 14.7; figuras 7.5 e 7.6 no Capítulo 7); os modelos de objeto descrevem as classes de objeto (Figura 14.2); os modelos de generalização ou herança (figuras 8.8, 8.9 e 8.10, Capítulo 8) mostram como as classes podem ser generalizações de outras classes; e os modelos de agregação (Figura 8.11) mostram como os conjuntos de objetos estão relacionados.

A Figura 14.10 mostra os objetos nos subsistemas da estação meteorológica. Também apresento algumas associações neste modelo. Por exemplo, o objeto **ControladorDeComunicações** está associado ao objeto **EstaçãoMeteorológica**, que por sua vez está associado ao package **ColetaDeDados**. Isso significa que esse objeto está associado a um ou mais objetos nesse package. Um modelo de packages mais um modelo de classes de objeto devem descrever os agrupamentos lógicos no sistema.

Um modelo de subsistemas é um modelo estático útil na medida em que mostra como o projeto pode ser organizado em grupos de objetos logicamente relacionados. Já usei esse tipo de modelo na Figura 14.6 para mostrar os subsistemas no sistema de mapeamento meteorológico. Os packages UML contêm construções encapsuladas e não são refletidos diretamente nas entidades do sistema desenvolvido. No entanto, eles podem refletir-se na estruturação de construções como bibliotecas em Java.

Os *modelos seqüenciais* são modelos dinâmicos que documentam, para cada modo de interação, a seqüência de interações que ocorrem entre objetos. A Figura 14.11 é um exemplo de um modelo de seqüência que mostra as operações envolvidas na coleta de dados de uma estação meteorológica. Em um modelo de seqüência:

1. Os objetos envolvidos na interação são organizados horizontalmente com uma linha vertical ligada a cada objeto.
2. O tempo é representado verticalmente de modo que progride para baixo nas linhas verticais tracejadas. Portanto, a seqüência de operações pode ser lida facilmente com base nesse modelo.
3. As setas rotuladas que ligam as linhas verticais representam as interações entre os objetos. Essas setas *não* são fluxos de dados, mas sim mensagens ou eventos fundamentais para as interações.
4. O retângulo estreito sobre a linha de vida do objeto representa o tempo em que este é o objeto de controle no sistema. Um objeto assume o controle no topo desse retângulo e passa o controle para um outro objeto na base do retângulo. Se houver uma hierarquia de chamadas, o controle não será passado até que o último retorno para a chamada inicial do método tenha sido completado.

Figura 14.10

Pacotes da estação meteorológica.

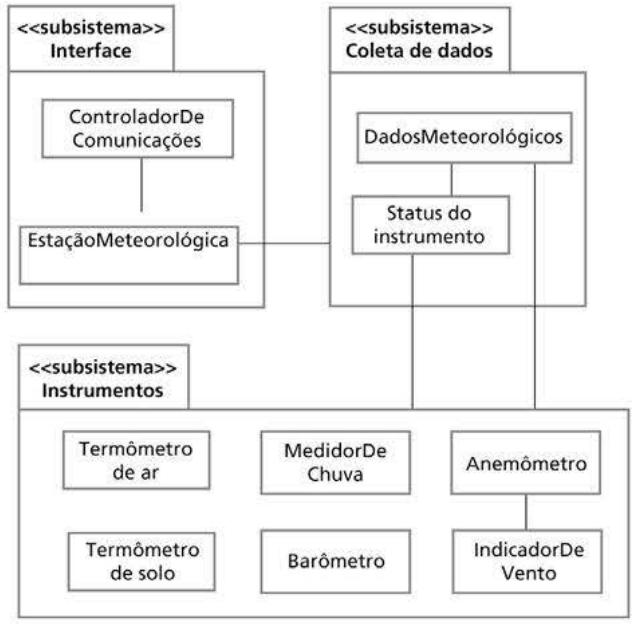
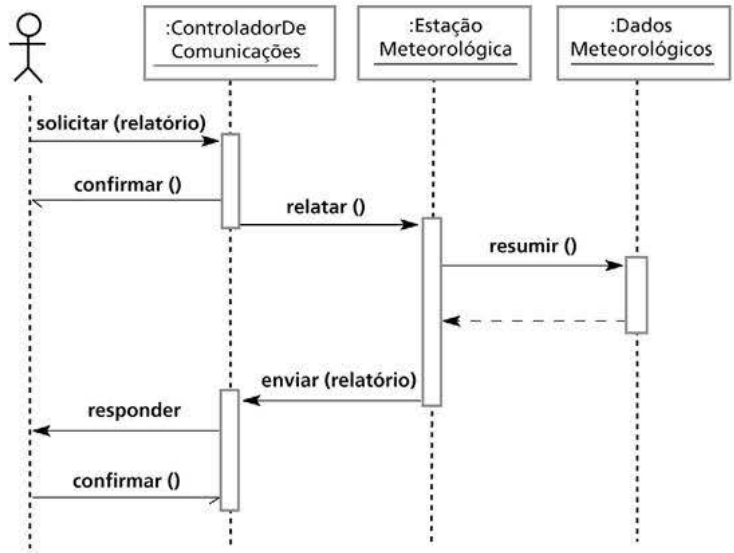


Figura 14.11

Seqüência de operações — coleta de dados.



Ao documentar um projeto, você deve produzir um modelo de seqüência para cada interação significativa. Se você desenvolveu um modelo de caso de uso, deve haver um modelo de seqüência para cada caso de uso identificado.

A Figura 14.11 mostra a seqüência de interações quando o sistema de mapeamento externo solicita os dados de uma estação meteorológica. Os diagramas de seqüência são lidos do topo para a base:

1. Um objeto que é uma instância de **ControladorDeComunicações** (:**ControladorDeComunicações**) recebe uma solicitação de seu ambiente para enviar um relatório meteorológico. Ele acusa o recebimento dessa solicitação. A meia ponta da seta da mensagem de reconhecimento (*acknowledge*) indica que o emissor da mensagem não espera uma resposta.
2. Esse objeto envia uma mensagem para um objeto que é uma instância de **EstaçãoMeteorológica** para criar um relatório meteorológico. A instância de **ControladorDeComunicações** suspende a si mesma (sua caixa de controle termina). O estilo de ponta de seta usado indica que as instâncias de objeto **ControladorDeComunicações** e **EstaçãoMeteorológica** são objetos que podem ser executados concorrentemente.
3. O objeto, que é uma instância de **EstaçãoMeteorológica**, envia uma mensagem para um objeto de **DadosMeteorológicos** para resumir os dados meteorológicos. Nesse caso, o estilo diferente de ponta de seta indica que a instância de **EstaçãoMeteorológica** espera uma resposta.
4. Este resumo é processado e o controle retorna para o objeto **EstaçãoMeteorológica**. A seta tracejada indica um retorno de controle.
5. Esse objeto envia uma mensagem ao **ControladorDeComunicações** solicitando-lhe que transfira os dados para o sistema remoto. O objeto **EstaçãoMeteorológica**, então, interrompe a si mesmo.
6. O objeto **ControladorDeComunicações** envia os dados resumidos ao sistema remoto, recebe uma confirmação (*acknowledge*) e, depois, interrompe a si mesmo aguardando a próxima solicitação.

A partir do diagrama de seqüência, podemos ver que o objeto **ControladorDeComunicações** e o objeto **EstaçãoMeteorológica** são realmente processos concorrentes, cuja execução pode ser interrompida e retomada. Essencialmente, a instância do objeto **ControladorDeComunicações** escuta as mensagens do sistema externo, decodifica essas mensagens e inicia as operações da estação meteorológica.

Os diagramas de seqüência são usados para modelar o comportamento combinado de um grupo de objetos, mas você pode querer resumir o comportamento de um único objeto em resposta às mensagens que pode processar. Para fazer isso, você pode usar um modelo de máquina de estado que mostra como a instância de objeto muda seu estado dependendo das mensagens recebidas. A UML usa *statecharts*, inventados por Harel (Harel, 1987), para descrever modelos de máquina de estado.

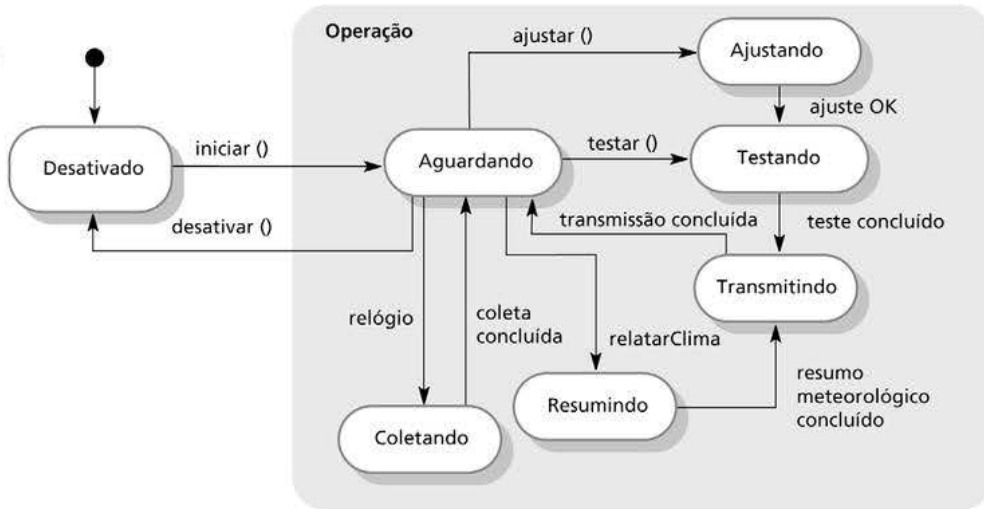
A Figura 14.12 é um statechart para o objeto **EstaçãoMeteorológica** que mostra como ele responde às solicitações de vários serviços.

Você pode ler esse diagrama da seguinte maneira:

1. Se o estado do objeto for **Desativado**, ele poderá somente responder à mensagem **iniciar()**. Ele passará, então, a um estado em que ficará aguardando outras mensagens. Uma seta não rotulada com o ponto preto indica que o estado **Desativado** é o estado inicial.

Figura 14.12

Diagrama de estado para EstaçãoMeteorológica.



2. No estado **Aguardando**, o sistema espera outras mensagens. Se uma mensagem **desativar()** for recebida, o objeto retornará ao estado **Desativado**.
3. Se uma mensagem **relatarClima()** for recebida, o sistema passará ao estado **Sumarizando**. Quando o sumário estiver concluído, o sistema passará a um estado **Transmitindo** no qual as informações são transmitidas por meio do **ControladorDeComunicações**. Ele retornará, então, ao estado **Aguardando**.
4. Se uma mensagem **calibrar()** for recebida, o sistema passará ao estado **Calibrando**, ao estado **Testando** e ao estado **Transmitindo**, antes de retornar ao estado **Aguardando**. Se a mensagem **testar**() for recebida, o sistema passará diretamente ao estado **Testando**.
5. Se um sinal do relógio for recebido, o sistema passará ao estado **Coletando**, no qual estará coletando dados dos instrumentos. Cada instrumento por sua vez será instruído a coletar seus dados.

Geralmente, você não precisa elaborar um *statechart* para todos os objetos definidos. Muitos objetos em um sistema são relativamente simples, e o modelo de máquina de estado não auxiliaria os implementadores a compreender esses objetos.

14.2.5 Especificação de interface entre objetos

Uma importante parte de qualquer projeto é a especificação de interfaces entre os componentes do projeto. Você precisa especificar as interfaces de modo que os objetos e os subsistemas possam ser projetados paralelamente. Após a especificação de uma interface, os desenvolvedores de outros objetos podem supor que essa interface será implementada.

A inclusão de detalhes de representação de interface deve ser evitada em um projeto de interface. A representação deve ser ocultada e as operações de objeto para acessar e atualizar os dados devem ser fornecidas. Se a representação estiver oculta, ela poderá ser alterada sem afetar os objetos que usam esses atributos. Isso leva a um projeto inerentemente mais fácil de manter. Por exemplo, uma representação de vetor de uma pilha pode ser alterada para uma representação de lista sem afetar outros objetos que usam a pilha. Por outro lado, muitas vezes faz sentido expor os atributos em um modelo estático de projeto, na medida em que essa é a maneira mais compacta de ilustrar as características essenciais dos objetos.

Não existe necessariamente um relacionamento 1:1 simples entre objetos e interfaces. O mesmo objeto pode ter várias interfaces, cada uma das quais sendo um ponto de vista sobre os métodos fornecidos. Essa condição apóia-se diretamente em Java, em que as interfaces são declaradas separadamente dos objetos e os objetos 'implementam' as interfaces. Da mesma maneira, todo um grupo de objetos pode ser acessado por meio de uma única interface.

O projeto de interface de objeto dedica-se à especificação de detalhes de interface de um objeto ou um grupo de objetos. Isso significa definir assinaturas e a semântica dos serviços fornecidos pelo objeto ou por um grupo de objetos. As interfaces podem ser especificadas em UML usando a mesma notação como em um diagrama de classes. Contudo, não existe seção de atributo e o estereótipo em UML <interface> deve ser incluído na parte do nome.

Uma abordagem alternativa, que particularmente prefiro, é usar uma linguagem de programação para definir a interface. Isso está ilustrado no Quadro 14.2, que mostra a especificação de interface da estação meteorológica em Java. À medida

Quadro 14.2

Descrição em Java da interface da estação meteorológica.

```
interface EstaçãoMeteorológica {

    public void EstaçãoMeteorológica () ;

    public void iniciar () ;
    public void iniciar (Instrumento i) ;

    public void desativar () ;
    public void desativar (Instrumento i) ;

    public void relatarClima ( ) ;

    public void testar () ;
    public void testar (Instrumento i) ;

    public void ajustar (Instrumento i) ;

    public int obterID () ;

} // EstaçãoMeteorológica
```

que as interfaces ficam mais complexas, essa abordagem torna-se mais eficiente, pois os recursos de verificação de sintaxe do compilador podem ser usados para descobrir erros e inconsistências na descrição da interface. A descrição em Java pode mostrar que alguns métodos podem apresentar diferentes números de parâmetros. Portanto, o método desativar pode ser aplicado à estação como um todo, caso não haja parâmetros, ou a um único instrumento.

14.3 Evolução de projeto

Depois que uma decisão foi tomada para desenvolver um sistema como um sistema de coleta de dados meteorológicos, é inevitável que sejam feitas propostas para mudanças no sistema. Uma vantagem importante de uma abordagem orientada a objetos para projeto é que ela simplifica o problema de efetuar mudanças no projeto. A razão para isso é que a representação de estado do objeto não influencia o projeto. Mudanças em detalhes internos de um objeto provavelmente não afetarão quaisquer outros objetos de sistema. Além disso, devido aos objetos não serem firmemente acoplados, a introdução de novos objetos sem efeitos significativos sobre o restante do sistema ocorre geralmente de maneira direta.

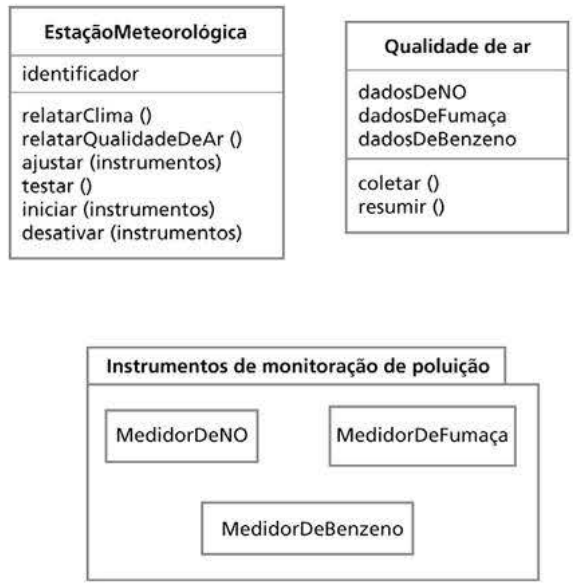
Para mostrar como uma abordagem orientada a objetos para projeto facilita as mudanças, considere que capacidades de monitoração de poluição devam ser acrescentadas a cada estação meteorológica. Isso envolve acrescentar um medidor de qualidade do ar para calcular a quantidade de diversos poluentes na atmosfera. As leituras de níveis de poluição são transmitidas simultaneamente com os dados meteorológicos. Para modificar o projeto, as seguintes mudanças devem ser feitas:

1. Uma classe de objeto chamada **QualidadeDoAr** deve ser introduzida como parte da **EstaçãoMeteorológica** no mesmo nível de **DadosMeteorológicos**.
2. Uma operação **relatarQualidadeDoAr** deve ser acrescentada à **EstaçãoMeteorológica** para enviar as informações de índices de poluição ao computador central. O software de controle da estação meteorológica deve ser modificado de modo que as leituras de índices de poluição sejam coletadas automaticamente quando solicitadas pelo objeto **EstaçãoMeteorológica** de alto nível.
3. Os objetos que representam os tipos de instrumentos de monitoração de índices de poluição devem ser acrescentados. Nesse caso, os níveis de óxido nitroso (NO), fumaça e benzeno podem ser medidos.

Os objetos de monitoração de níveis de poluição são encapsulados em um package separado chamado **Instrumentos de Monitoração de Poluição**. Esse package tem associações com **Qualidade do Ar e EstaçãoMeteorológica**, mas não com quaisquer objetos usados para coletar dados meteorológicos. A Figura 14.13 mostra a **EstaçãoMeteorológica** e os novos objetos acrescentados ao sistema. Com exceção do nível mais alto do sistema (**EstaçãoMeteorológica**), nenhuma mudança de software é necessária nos objetos originais da estação meteorológica. O acréscimo de coleta de dados de níveis de poluição não afeta, de modo algum, a coleta de dados meteorológicos.

Figura 14.13

Novos objetos para apoiar a monitoração de índices de poluição.



PONTOS-CHAVE

- O projeto orientado a objetos é uma abordagem para projeto de software na qual os componentes fundamentais do projeto representam objetos com seus estados privativos, bem como representam operações em vez de funções.
- Um objeto deve ter operações de construção e de inspeção que permitam a seu estado ser verificado e modificado. O objeto fornece serviços (operações que usam informações de estado) para outros objetos. Os objetos são criados em tempo de execução usando uma especificação contida em uma definição de classe de objeto.
- Os objetos podem ser implementados seqüencial ou concorrentemente. Um objeto concorrente pode ser um objeto passivo, cujo estado é alterado somente por meio de sua interface, ou um objeto ativo, que pode mudar seu próprio estado sem intervenção externa.
- A linguagem de modelagem unificada (UML — *Unified Modeling Language*) fornece uma variedade de notações que podem ser usadas para documentar um projeto orientado a objetos.
- O processo de projeto orientado a objetos inclui atividades para projetar a arquitetura de sistema, identificar objetos de sistema, descrever o projeto usando modelos de objeto diferentes e documentar as interfaces de objeto.
- Vários modelos diferentes podem ser produzidos durante um processo de projeto orientado a objetos. Esses modelos incluem modelos estáticos (modelos de classe, modelos de generalização, modelos de associação) e modelos dinâmicos (modelos de seqüência, modelos de máquina de estado).
- As interfaces com o objeto devem ser definidas precisamente de modo que outros objetos possam usá-la. Uma linguagem de programação como Java pode ser usada para documentar as interfaces com o objeto.
- Uma importante vantagem do projeto orientado a objetos é que ele simplifica a evolução do sistema.

LEITURAS SUGERIDAS

Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process, 2nd ed. Uma boa introdução ao uso da UML dentro de um processo de projeto orientado a objetos. Sua abrangência de padrões de projeto é também uma leitura relevante para o Capítulo 18. (C. Larman, 2001, Prentice Hall.)

The unified modeling language user guide. Um texto definitivo sobre UML e seu uso para descrição de projetos orientados a objetos. Existem dois textos associados — um deles é um manual de consulta de UML e outro propõe um processo de desenvolvimento orientado a objetos. (G. Booch, et al., 1999, Addison-Wesley.)

Um novo padrão para a UML (UML 2.0) estava pronto em meados de 2003, mas, no momento em que este livro estava sendo escrito, esses livros não tinham sido atualizados para refletir esse novo padrão.

Existe também uma quantidade imensa de material introdutório e de tutoriais sobre UML na Web. Inclui alguns links nas páginas Web do livro.

EXERCÍCIOS

14.1 Explique por que a adoção de uma abordagem para projeto baseada em objetos não firmemente acoplados, que ocultam informações sobre sua representação, pode levar a um projeto que pode ser prontamente modificado.

14.2 Usando exemplos, explique a diferença entre um objeto e uma classe de objeto.

14.3 Sob quais circunstâncias você pode desenvolver um projeto em que os objetos são executados concorrentemente?

14.4 Usando a notação gráfica da UML para classes de objeto, projete as seguintes classes identificando atributos e operações. Use sua própria experiência para decidir sobre os atributos e as operações que devem ser associados a esses objetos:

- um telefone;
- uma impressora para um computador pessoal;
- um sistema de som;
- uma conta bancária; e
- um catálogo de biblioteca.

14.5 Desenvolva o projeto da estação meteorológica detalhadamente, propondo descrições de interface com objetos mostrados na Figura 14.9. Essas descrições podem ser expressas em Java, C++ ou UML.

14.6 Desenvolva o projeto da estação meteorológica para mostrar a interação entre o subsistema de coleta de dados e os instrumentos que coletam dados meteorológicos. Use diagramas de seqüência para mostrar essa interação.

14.7 Identifique possíveis objetos nos seguintes sistemas e desenvolva um projeto orientado a objetos para eles. Você pode formular quaisquer hipóteses razoáveis sobre os sistemas ao derivar o projeto.

- Um sistema de gerenciamento de agendas e de horários em grupo tem a intenção de apoiar os horários de reuniões e compromissos de um grupo de funcionários. Quando um compromisso que envolve certo número de pessoas deve ser marcado, o sistema encontra um horário comum em cada uma das agendas e marca o compromisso para esse horário. Se nenhum horário estiver disponível, o sistema interage com os usuários para reorganizar as agendas pessoais e procurar espaço para o compromisso.
- Um posto de combustível deve ser definido para operação completamente automatizada. Os motoristas passam seus cartões de crédito em uma leitora conectada à bomba; o cartão é verificado comunicando-se com um computador da administradora de cartões de crédito e um limite de combustível é estabelecido. O motorista pode, então, obter o combustível necessário. Quando o abastecimento de combustível é concluído e a mangueira é colocada de volta no suporte da bomba, a conta do cartão de crédito do motorista é debitada com o valor do combustível utilizado. O cartão de crédito é devolvido depois da operação de débito. Se o cartão for inválido, a bomba devolverá o cartão antes de o combustível ser enviado ao tanque.

14.8 Escreva definições de interface precisas em Java ou C++ para os objetos definidos no Exercício 14.7.

14.9 Elabore um diagrama de seqüência mostrando as interações de objetos em um sistema de agenda em grupo quando um grupo de pessoas marca uma reunião.

14.10 Elabore um statechart que mostre as possíveis mudanças de estado em um ou mais objetos definidos no Exercício 14.7.

15

Projeto de software de tempo real

Objetivos

Os objetivos deste capítulo são apresentar as técnicas usadas no projeto de sistemas de tempo real e descrever algumas arquiteturas genéricas nesses sistemas. Após ler este capítulo, você:

- compreenderá o conceito de um sistema de tempo real e por que os sistemas de tempo real geralmente são implementados como um conjunto de processos concorrentes;
- terá sido apresentado a um processo de projeto para sistemas de tempo real;
- compreenderá o papel de um sistema operacional de tempo real;
- conhecerá as arquiteturas genéricas de processo para sistemas de monitoração e controle e sistemas de aquisição de dados.

Conteúdo

15.1 Projeto de sistemas
15.2 Sistemas operacionais de tempo real
15.3 Sistemas de monitoração e controle
15.4 Sistemas de aquisição de dados

Os computadores são usados para controlar uma ampla variedade de sistemas, desde máquinas domésticas simples até fábricas inteiras. Estes computadores interagem diretamente com dispositivos de hardware. O software nesses sistemas é um *software embutido de tempo real* que deve reagir a eventos gerados pelo hardware e emitir sinais de controle em resposta a esses eventos. Ele está embutido em alguns sistemas de hardware de grande porte e deve responder, em tempo real, aos eventos do ambiente do sistema.

Os sistemas embutidos de tempo real são diferentes de outros tipos de sistemas de software. Seu funcionamento correto depende da capacidade de resposta do sistema aos eventos dentro de um curto intervalo de tempo. Defino um sistema de tempo real da seguinte maneira:

Um sistema de tempo real é um sistema de software cujo funcionamento correto depende dos resultados produzidos pelo sistema e do tempo em que estes resultados são produzidos. Um

sistema de tempo real leve é aquele cuja operação será degradada caso os resultados não sejam produzidos de acordo com os requisitos de timing especificados. Um sistema de tempo real rígido é aquele cuja operação será incorreta se os resultados não forem produzidos de acordo com a especificação de timing.

A resposta no tempo correto é um fator importante em todos os sistemas embutidos, mas, em alguns casos, respostas muito rápidas não são necessárias. Por exemplo, o sistema de bomba de insulina, que uso como exemplo em vários capítulos neste livro, é um sistema embutido. Contudo, enquanto ele necessita verificar o nível de glicose em intervalos periódicos, não necessita responder de maneira muito rápida aos eventos externos. Portanto, uso exemplos diferentes neste capítulo para ilustrar os fundamentos do projeto de sistemas de tempo real.

Uma maneira de olhar um sistema de tempo real é como um sistema de estímulo–resposta. Dado um estímulo entrada, o sistema deve produzir uma resposta correspondente. O comportamento de um sistema de tempo real, portanto, pode ser definido por meio de uma lista de estímulos recebidos pelo sistema, das respostas associadas e do tempo em que a resposta deve ser produzida.

Os estímulos dividem-se em duas categorias:

1. *Estímulos periódicos*. Esses estímulos ocorrem em intervalos de tempo previsíveis. Por exemplo, o sistema pode examinar um sensor a cada 50 milissegundos e reagir (responder) de acordo com o valor desse sensor (o estímulo).
2. *Estímulos aperiódicos*. Esses estímulos ocorrem irregularmente. São geralmente sinalizados por meio do mecanismo de interrupção do computador. Um exemplo de tal estímulo seria uma interrupção que indique que uma transferência de E/S (entrada/saída) foi concluída e os dados estão disponíveis em um *buffer*.

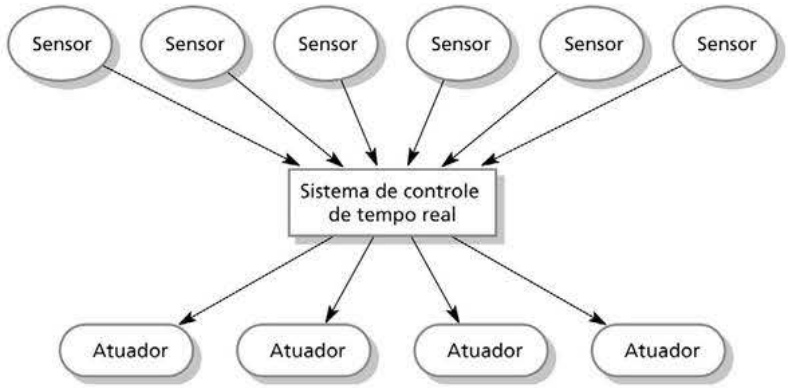
Os estímulos periódicos em um sistema de tempo real são normalmente gerados por sensores associados ao sistema. Eles fornecem informações sobre o estado do ambiente do sistema. As respostas são encaminhadas a um conjunto de atuadores que controlam algum equipamento, como uma bomba, que influencia o ambiente do sistema. Os estímulos aperiódicos podem ser gerados tanto pelos atuadores como pelos sensores. Eles indicam, muitas vezes, alguma condição excepcional, como uma falha de hardware a ser tratada pelo sistema. Esse modelo sensor–sistema–atuador de um sistema embutido de tempo real é ilustrado na Figura 15.1.

Um sistema de tempo real tem de responder aos estímulos que ocorrem em tempos diferentes. Sua arquitetura deve, portanto, ser organizada de modo que, tão logo um estímulo seja recebido, o controle seja transferido para o tratador correto. Isso não é prático em programas seqüenciais. Sendo assim, os sistemas de tempo real normalmente são projetados como um conjunto de processos concorrentes que cooperam entre si. Para apoiar o gerenciamento desses processos, a plataforma de execução da maioria dos sistemas de tempo real inclui um *sistema operacional de tempo real*. Os recursos, nesse sistema operacional, são acessados por meio de um sistema de apoio de tempo real para a linguagem de programação de tempo real usada.

A generalidade desse modelo de estímulo–resposta de um sistema de tempo real conduz a um modelo de arquitetura genérica e abstrata, no qual existem três tipos de processos. Para cada tipo de sensor existe um processo de gerenciamento de sensores; os processos computacionais calculam a resposta exigida para os estímulos recebidos pelo sistema; os processos de controle de atuadores gerenciam a operação de atuadores. Esse modelo permite que os dados sejam coletados rapidamente do sensor (antes de a próxima entrada tornar-se disponível) e que o processamento e a resposta do atuador associado sejam realizados posteriormente.

Essa arquitetura genérica pode ser instanciada em uma série de arquiteturas de aplicações diferentes que ampliam o conjunto de arquiteturas explicadas no Capítulo 13. As arquiteturas de aplicações de tempo real são instâncias de arquitetura dirigida a eventos nas quais os estímulos, direta ou indiretamente, causam a geração de eventos. Neste capítulo, apresento duas arquiteturas de aplicações adicionais: a arquitetura para sistemas de monitoração e controle (na Seção 15.3), e a arquitetura para sistemas de aquisição de dados (na Seção 15.4).

Figura 15.1
Modelo geral de um sistema de tempo real.



As linguagens de programação para desenvolvimento de sistemas de tempo real precisam incluir recursos para acessar o hardware do sistema e deve ser possível prever o timing de operações específicas nessas linguagens. Sistemas de tempo real rígidos são ainda, algumas vezes, programados em linguagem assembly de modo que os prazos finais (*deadlines*) rígidos possam ser cumpridos. As linguagens de nível de sistema, como C, que permitem que códigos eficientes sejam gerados, são também amplamente usadas.

A vantagem do uso de uma linguagem de programação de sistemas de baixo nível, como C, é que ela permite o desenvolvimento de programas muito eficientes. Contudo, essas linguagens não incluem estruturas para apoiar a simultaneidade ou o gerenciamento de recursos compartilhados. Esses recursos são implementados por meio de chamadas ao sistema operacional de tempo real que não podem ser verificadas pelo compilador; sendo assim, os erros de programação são mais prováveis. Os programas muitas vezes são difíceis de compreender devido às características de tempo real não serem explícitas no programa.

Nos últimos anos, tem havido trabalho extensivo para ampliar a linguagem Java para desenvolvimento de sistemas de tempo real (Nilsen, 1998; Higuera-Toledano e Issarny, 2000; Hardin, et al., 2002). Esse trabalho envolve a modificação da linguagem para resolver problemas fundamentais de tempo real:

1. Não é possível especificar o momento em que os threads devem ser executados.
2. A coleta de lixo não é controlável — pode ser iniciada a qualquer momento. Portanto, o timing de comportamento dos threads é imprevisível.
3. Não é possível descobrir os tamanhos das filas associadas aos recursos compartilhados.
4. A implementação da Máquina Virtual Java varia de um computador para outro; sendo assim, o mesmo programa pode ter timing de comportamento diferente.
5. A linguagem não permite a análise detalhada de espaço de run-time ou processador.
6. Não existem modos-padrão de acesso ao hardware do sistema.

As versões de tempo real em Java, como J2ME (Java 2 Micro Edition) da Sun, estão disponíveis atualmente. Uma série de fornecedores fornece implementações de Máquina Virtual Java adaptadas ao desenvolvimento de sistemas de tempo real. Esses progressos significam que o Java se tornará cada vez mais usado como linguagem de programação de tempo real.

15.1 Projeto de sistemas

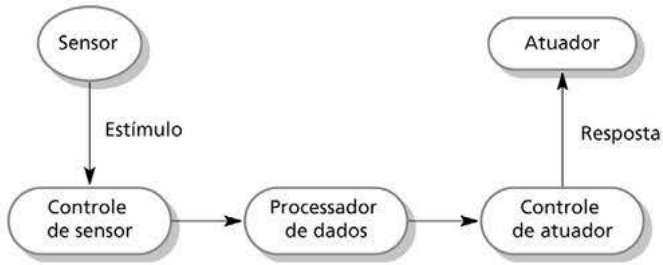
Conforme explicado no Capítulo 2, parte do processo de projeto de sistema envolve a decisão sobre quais capacidades do sistema serão implementadas no software e no hardware. Para muitos sistemas de tempo real embutidos em produtos para o consumidor, como os sistemas em telefones celulares, os custos e o consumo de energia do hardware são críticos. Podem ser usados processadores específicos projetados para apoiar os sistemas embutidos e, em alguns sistemas, pode haver necessidade de se projetar e construir um hardware de propósito especial.

Isso significa que um processo de projeto *top-down* — no qual o projeto começa com um modelo abstrato decomposto e desenvolvido em uma série de estágios — não é prático para a maioria dos sistemas de tempo real. As decisões de baixo nível sobre o hardware, software de apoio e timing do sistema devem ser consideradas no início do processo. Essas decisões limitam a flexibilidade dos projetistas de sistemas e podem significar que uma funcionalidade adicional seja necessária, como o gerenciamento de bateria e energia.

Os eventos (os estímulos) no lugar de objetos ou funções devem ser a parte central do processo de projeto de software de tempo real. Existem vários estágios intercalados nesse processo de projeto:

1. Identificar os estímulos que o sistema deve processar e as respostas associadas.
2. Para cada estímulo e resposta associada, identificar as restrições de timing que se aplicam a ambos, estímulo e processamento de resposta.
3. Escolher uma plataforma de execução para o sistema: o hardware e o sistema operacional de tempo real a serem usados. Os fatores que influenciam essas escolhas incluem as restrições de *timing* do sistema, limitações de energia disponível, a experiência da equipe de desenvolvimento e o preço-alvo para o sistema a ser entregue.
4. Agregar o estímulo e o processamento de resposta em uma série de processos concorrentes. Uma boa regra prática em projeto de sistemas de tempo real é associar um processo a cada categoria de estímulo e resposta, conforme mostrado na Figura 15.2.
5. Para cada estímulo e resposta, projetar algoritmos para realizar os processamentos necessários. Os projetos de algoritmos precisam ser, muitas vezes, desenvolvidos relativamente no início do processo de projeto para indicar a quantidade de processamento e o tempo exigidos para completar esse processamento.
6. Projetar um sistema de cronograma que assegurará que os processos sejam iniciados a tempo de cumprir seus prazos.

Figura 15.2
Processos de controle de sensor–atuador.



A ordem dessas atividades no processo depende do tipo de sistema que está sendo desenvolvido e seus requisitos de processo e de plataforma. Em alguns casos, você pode ser capaz de seguir uma abordagem bastante abstrata, começando com os estímulos e o processamento associado, e tomando decisões sobre o hardware e as plataformas de execução posteriormente no processo. Em outros casos, a escolha do hardware e do sistema operacional é feita antes que o projeto de software comece, sendo necessário orientar o projeto com base nas capacidades de hardware.

Os processos em um sistema de tempo real precisam ser coordenados. Os mecanismos de coordenação de processo asseguram exclusão mútua a recursos compartilhados. Quando um processo está modificando um recurso compartilhado, outros processos não devem ser capazes de alterar esse recurso. Os mecanismos para assegurar a exclusão mútua incluem semáforos (Dijkstra, 1968), monitores (Hoare, 1974) e regiões críticas (Brinch-Hansen, 1973). Esses mecanismos são descritos na maioria dos textos sobre sistemas operacionais (Tanenbaum, 2001; Silberschatz, et al., 2002).

Após escolher uma plataforma de execução para o sistema, projetar uma arquitetura de processo e decidir sobre uma política de cronograma, pode ser necessário verificar se o sistema atenderá aos seus requisitos de timing. Você pode fazer isso por meio de análise estática do sistema usando conhecimentos de como os componentes se comportam em relação ao timing ou por meio de simulação. Essa análise pode revelar que o sistema não funcionará adequadamente. A arquitetura de processo, a política de cronograma, a plataforma de execução ou todos esses elementos podem ter necessidade de ser reprojetados para aprimorar o desempenho do sistema.

A análise de timing em sistemas de tempo real é difícil. Devido aos estímulos aperiódicos serem imprevisíveis, os projetistas precisam formular hipóteses sobre a probabilidade de ocorrência desses estímulos (e, portanto, solicitar serviços) em qualquer momento específico. Essas hipóteses podem estar incorretas e o desempenho do sistema após a entrega pode não ser adequado. O livro de Cooling (Cooling, 2003) explica técnicas para análise de desempenho do sistema de tempo real.

Como os sistemas de tempo real devem atender aos seus requisitos de timing, pode não ser possível usar o desenvolvimento orientado a objetos para sistemas de tempo real rígidos. O desenvolvimento orientado a objetos envolve a ocultação de representações de dados e o acesso aos valores de atributos por meio de operações definidas com o objeto. Isso significa que existe um overhead de desempenho significativo em sistemas orientados a objetos, pois é necessário um código extra para intermediar o acesso aos atributos e tratar as chamadas para as operações. A conseqüente perda de desempenho pode tornar impossível atender aos prazos de tempo real.

As restrições de timing ou outros requisitos podem, algumas vezes, significar que é melhor implementar algumas funções de sistema, como processamento de sinais, em hardware no lugar de software. Os componentes de hardware proporcionam desempenho muito melhor do que o software equivalente. Os gargalos de processamento do sistema podem ser identificados e substituídos por hardware, evitando, assim, a otimização dispendiosa de software.

15.1.1 Modelagem de sistemas de tempo real

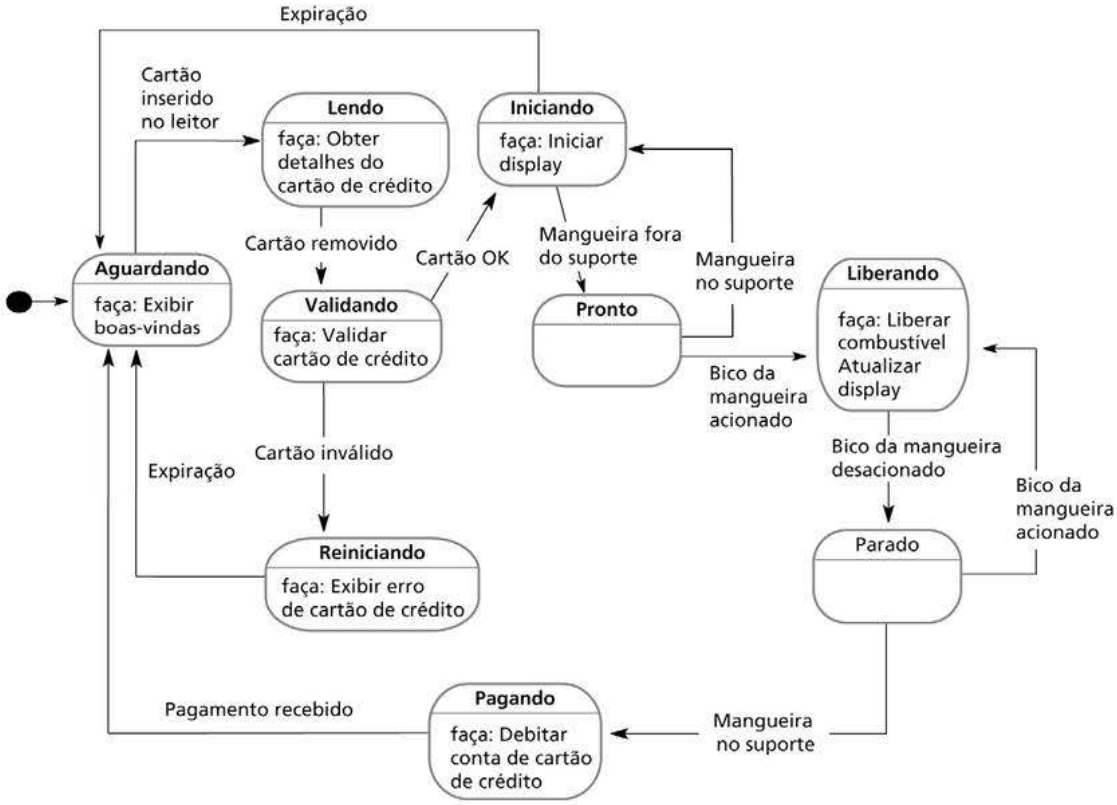
Os sistemas de tempo real precisam responder aos eventos que ocorrem em intervalos irregulares. Esses eventos (ou estímulos) causam, muitas vezes, a passagem do sistema para um estado diferente. Por essa razão, a modelagem de máquina de estado, descrita no Capítulo 8, é freqüentemente usada para modelar sistemas de tempo real.

Os modelos de máquinas de estado são uma boa maneira, independente de linguagem, de representar um projeto de sistema de tempo real e, portanto, são parte integrante dos métodos de projeto de sistema de tempo real (Gomma, 1993). A UML apóia o desenvolvimento de modelos de estado baseados em *statecharts* (Harel, 1987; Harel, 1988). Os *statecharts* estruturam os modelos de estado de modo que grupos de estados podem ser considerados como uma única entidade. Douglass explica o uso da UML em desenvolvimento de sistemas de tempo real (Douglass, 1999).

Um modelo de estado de um sistema considera que, a qualquer momento, o sistema está em um dos vários estados possíveis. Quando um estímulo for recebido, isso pode causar a transição para um estado diferente. Por exemplo, um sistema que controla uma válvula pode se mover de um estado 'Válvula aberta' para um estado 'Válvula fechada' quando um comando de operador (estímulo) for recebido.

Figura 15.3

Modelo de máquina de estado de uma bomba de combustível.



Já ilustrei essa abordagem para modelagem de sistemas no Capítulo 8 usando o modelo de um forno de microondas simples. A Figura 15.3 é um exemplo de modelo de máquina de estados que mostra a operação de um sistema de software de liberação de combustível embutido em uma bomba de gasolina. Os retângulos com cantos arredondados representam os estados de sistema e as setas representam os estímulos que forçam a transição de um estado para outro. Os nomes escolhidos nesse diagrama de máquina de estado são descritivos e as informações associadas indicam as ações tomadas pelos atuadores do sistema ou as informações exibidas.

O sistema de liberação de combustível é projetado para permitir a operação sem atendentes. O comprador insere um cartão de crédito em um leitor de cartões montado na bomba. Isso causa a transição para o estado **Lendo**, no qual os detalhes do cartão são lidos e é solicitado que o comprador retire o cartão. O sistema passa ao estado **Validando**, no qual o cartão é validado. Se o cartão for válido, o sistema inicializará a bomba e, quando a mangueira de combustível for removida de seu suporte, ela estará pronta para liberar o combustível. Ativar o gatilho sob o bico da mangueira causa o bombeamento do combustível; o bombeamento pára quando o gatilho é liberado (para simplificar, ignorei a chave de pressão projetada para interromper o derramamento de combustível). Depois que a liberação de combustível for finalizada e o comprador recolocar a mangueira em seu suporte, o sistema passará a um estado **Pagando**, no qual a conta do usuário é debitada.

15.2 Sistemas operacionais de tempo real

Quase todos os sistemas embutidos mais simples funcionam, atualmente, em conjunto com um sistema operacional de tempo real (RTOS — Real Time Operating System). Um sistema operacional de tempo real gerencia os processos e a alocação de recursos em um sistema de tempo real. Ele inicia e interrompe os processos, de modo que os estímulos possam ser tratados, e aloca memória e recursos de processador. Existem muitos produtos de RTOS disponíveis, desde sistemas simples e muito pequenos para dispositivos voltados ao consumidor até sistemas complexos para telefones celulares e dispositivos móveis, além de sistemas operacionais especialmente projetados para controle de processos e telecomunicações.

Os componentes de um RTOS (Figura 15.4) dependem do tamanho e da complexidade do sistema de tempo real que está sendo desenvolvido. Em todos eles, exceto nos sistemas mais simples, geralmente há:

1. *Um relógio de tempo real* que fornece informações para programar processos periodicamente.
2. *Um tratador de interrupções* que gerencia as solicitações aperiódicas de serviços.
3. *Um programador*. Esse componente é responsável por examinar os processos que podem ser executados e por escolher um deles para execução.
4. *Um gerenciador de recursos*. Dado um processo programado para execução, o gerenciador de recursos aloca memória apropriada e recursos de processador.
5. *Um despachador*. Esse componente é responsável por iniciar a execução de um processo.

Os sistemas operacionais de tempo real para sistemas de grande porte, como sistemas de controle de processos e de telecomunicações, podem ter recursos adicionais, como recursos de gerenciamento de armazenamento de disco e de gerenciamento de defeitos, que detectam e reportam defeitos de sistema, além de um gerenciador de configuração que apóia a reconfiguração dinâmica das aplicações de tempo real.

15.2.1 Gerenciamento de processos

Os sistemas de tempo real precisam tratar rapidamente os eventos externos e, em alguns casos, atender aos prazos de processamento desses eventos. Isso significa que os processos de tratamento de eventos devem ser programados para execução a tempo de detectar um evento e devem ser alocados recursos de processador suficientes para cumprir o prazo. O gerenciador de processos em um RTOS é responsável por escolher os processos para execução, alocar os recursos de processador e memória, e iniciar e interromper a execução de processos em um processador.

O gerenciador de processos precisa gerenciar processos com prioridades diferentes. Para alguns estímulos, como aqueles associados a certos eventos excepcionais, é essencial que o processamento seja concluído dentro dos limites de tempo especificados. Outros processos podem ser atrasados de modo seguro, caso um processo mais crítico solicite um serviço. Conseqüentemente, o RTOS precisa ser capaz de gerenciar pelo menos dois níveis de prioridade para processos do sistema:

1. *Nível de interrupção*. Este é o nível de prioridade mais alto. É alocado aos processos que necessitam de uma resposta muito rápida. Um desses processos será o processo de relógio de tempo real.
2. *Nível de relógio*. Esse nível de prioridade é alocado aos processos periódicos.

Figura 15.4

Componentes de um sistema operacional de tempo real.

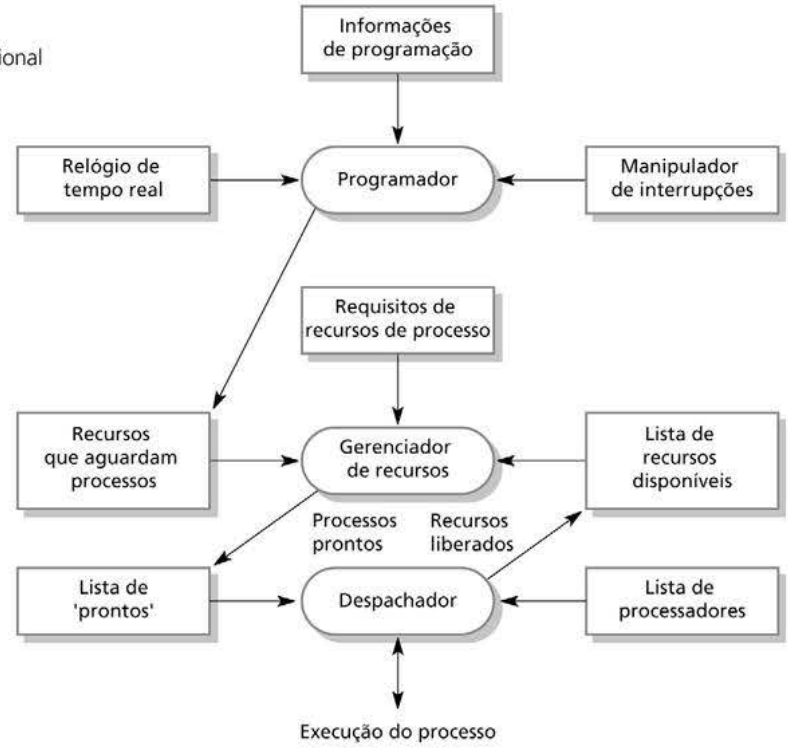
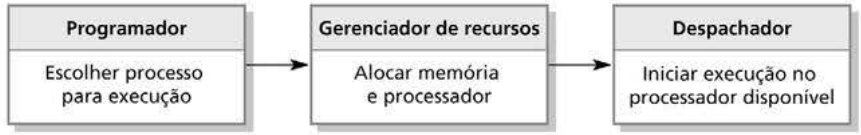


Figura 15.5

Ações de RTOS necessárias para iniciar um processo.



Pode haver um nível de prioridade adicional alocado aos processos de segundo plano (como um processo de autoteste) que não necessita atender aos prazos de tempo real. Esses processos são programados para execução quando houver capacidade de processador disponível.

Dentro de cada um dos níveis de prioridade, diferentes categorias de processos podem ser alocadas com diferentes prioridades. Por exemplo, pode haver diversas linhas de interrupção. A interrupção de um dispositivo muito rápido pode ter de parar o processamento da interrupção de um dispositivo mais lento para evitar perda de informações. A alocação de prioridades de processo, de modo que todos os processos sejam atendidos a tempo, geralmente requer ampla análise e simulação.

Os processos periódicos devem ser executados em intervalos de tempo especificados para aquisição de dados e controle de atuadores. Na maioria dos sistemas de tempo real, haverá vários tipos de processos periódicos. Esses processos terão períodos (o tempo entre execuções de processo), tempos de execução e prazos (o tempo no qual o processamento deve ser concluído) diferentes. Ao usar os requisitos de timing especificados no programa da aplicação, o RTOS organiza a execução de processos periódicos de modo que todos eles possam atender aos seus prazos.

As ações tomadas pelo sistema operacional para gerenciamento de processos periódicos são mostradas na Figura 15.5. O programador examina a lista de processos periódicos e seleciona um processo a ser executado. A escolha depende da prioridade do processo, dos períodos de processo, dos tempos de execução esperados e dos prazos dos processos prontos. Algumas vezes, processos com prazos diferentes devem ser executados no mesmo tique do relógio. Em tal situação, um processo deve ser atrasado, desde que seu prazo possa ainda ser cumprido.

Os processos precisam responder aos eventos assíncronos dirigidos em geral a interrupções. O mecanismo de interrupção do computador causa a transferência do controle para uma posição na memória predeterminada. Essa posição contém uma instrução de jump para uma rotina simples e rápida de serviço de interrupção. Primeiro, a rotina de serviço desativa as outras interrupções para evitar ser interrompida por ela mesma. Ela descobre a causa da interrupção e inicia, com uma prioridade alta, um processo de tratamento do estímulo que causou a interrupção. Em alguns sistemas de aquisição de dados de alta velocidade, o tratador de interrupções salva os dados que a interrupção indicou como disponíveis em um buffer para processamento posterior. As interrupções são, em seguida, ativadas novamente e o controle é retornado ao sistema operacional.

A qualquer momento pode haver vários processos com prioridades diferentes que poderiam ser executados. O programador de processos implementa as políticas de programação de sistemas que determinam a ordem de execução dos processos. Existem duas estratégias básicas de programação:

1. *Programação não preemptiva*. Uma vez que um processo foi programado para execução, ele será executado até a conclusão ou até que seja interrompido por algum motivo, como a espera por uma entrada. Entretanto, isso pode causar problemas quando existirem processos com prioridades diferentes e um processo de prioridade alta tiver de aguardar o término de um processo de prioridade baixa.
2. *Programação preemptiva*. A execução de um processo pode ser interrompida se um processo de prioridade mais alta solicitar o serviço. O processo de prioridade mais alta interrompe a execução do processo de prioridade mais baixa e é alocado a um processador.

Dentro dessas estratégias, algoritmos de programação diferentes foram desenvolvidos. Essas estratégias incluem a programação *round-robin*, na qual cada processo é executado por vez, o escalamento *rate monotonic*, em que o processo com o período mais curto recebe prioridade maior e o prazo mais curto é programado para ocorrer primeiro (Burns e Wellings, 2001).

As informações sobre o processo a ser executado são passadas ao gerenciador de recursos. O gerenciador de recursos aloca memória e um processador para esse processo caso o sistema tenha vários processadores. O processo é, então, colocado na lista de 'prontos', uma lista de processos prontos para execução. Quando um processador conclui a execução de um processo e torna-se disponível, um despachador será invocado. Ele fará a leitura da lista de 'prontos' buscando um processo que possa ser executado no processador disponível e iniciará sua execução.

15.3 Sistemas de monitoração e controle

Os sistemas de monitoração e controle são uma categoria importante de sistemas de tempo real. Eles verificam sensores que fornecem informações sobre o ambiente do sistema e executam ações dependendo da leitura do sensor. Os

sistemas de monitoração executam ações quando algum valor excepcional de sensor for detectado. Os sistemas de controle controlam continuamente os atuadores de hardware dependendo do valor dos sensores associados.

A arquitetura característica dos sistemas de monitoração e controle é mostrada na Figura 15.6. Cada tipo de sensor que está sendo monitorado tem seu próprio processo de monitoração, assim como cada tipo de atuador que está sendo controlado. O processo de monitoração coleta e integra os dados antes de passá-los a um processo de controle, que toma decisões baseadas nesses dados e envia comandos de controle apropriados aos processos de controle de equipamentos. Em sistemas simples, as responsabilidades de monitoração e controle podem ser integradas em um único processo. Também mostrei dois outros processos que podem ser incluídos em sistemas de monitoração e controle. Esses processos são um processo de teste, que pode executar programas de teste de hardware, e um processo de painel de controle, que gerencia os painéis de controle do sistema ou console do operador.

Para ilustrar os sistemas de monitoração e controle, uso o exemplo de um sistema de alarme contra roubos que poderia ser instalado em um edifício de escritórios:

> Um sistema de software deve ser implementado para controlar um sistema de alarme contra roubos a ser instalado em edifícios comerciais. Esse sistema usa tipos diferentes de sensores. Esses sensores incluem detectores de movimento em salas individuais, sensores de janela nas janelas do andar térreo, que detectam a quebra de uma janela, e sensores de porta, que detectam a abertura das portas dos corredores. Existem 50 sensores de janela, 30 sensores de porta e 200 detectores de movimento no sistema.
>
> Quando um sensor detecta a presença de um intruso, o sistema chama automaticamente a polícia local e, com um sintetizador de voz, relata a posição do alarme. Ele acende as luzes nas salas ao redor do sensor ativado e aciona um alarme sonoro. O sistema de alarme é normalmente alimentado pelo sistema de energia central, mas é equipado com uma bateria de back-up. A perda de energia é detectada por meio de um monitor de circuito de energia separado que monitora a tensão principal. Ele interrompe o sistema de alarme caso uma queda de tensão seja detectada.

Esse sistema é um sistema de tempo real 'leve' que não tem requisitos de timing rigorosos. Os sensores não precisam detectar eventos de alta velocidade e, assim, eles precisam ser lidos somente duas vezes por segundo. Para tornar o exemplo mais fácil de compreender, simplifiquei o projeto deixando de fora os processos de teste e de display.

O processo de projeto segue as etapas explicadas na Seção 15.1; sendo assim, você deve começar pela identificação dos estímulos aperiódicos que o sistema recebe e as respostas associadas. Devido às simplificações de projeto propostas, os estímulos gerados pelos procedimentos de teste de sistema e os sinais externos para desligá-lo no evento de um alarme falso podem ser ignorados. Isso significa que existem somente duas categorias de estímulos a serem processados:

1. *Falha de energia*. Esse estímulo é gerado pelo monitor de circuito. A resposta necessária é alternar o circuito para a energia de back-up emitindo um sinal para um dispositivo de chaveamento de energia.
2. *Alarme de intrusos*. Esse estímulo é gerado por um dos sensores do sistema. A resposta para esse estímulo é verificar o número da sala do sensor ativado, fazer uma chamada para a polícia, ativar o sintetizador de voz para gerenciar a chamada e ligar o alarme sonoro contra intrusos e as luzes do edifício na área.

O próximo passo no processo de projeto é considerar as restrições de timing de cada estímulo e a resposta associada. Essas restrições de timing são mostradas na Tabela 15.1. Você, deve listar normalmente as restrições de timing de cada categoria de sensor separadamente, mesmo quando, como neste caso, eles forem os mesmos. Ao tratá-los separadamente, você deixa espaço para mudanças futuras e facilita o cálculo do número de vezes que o processo de controle precisa ser executado a cada segundo.

Figura 15.6

Arquitetura genérica para um sistema de monitoração e controle.

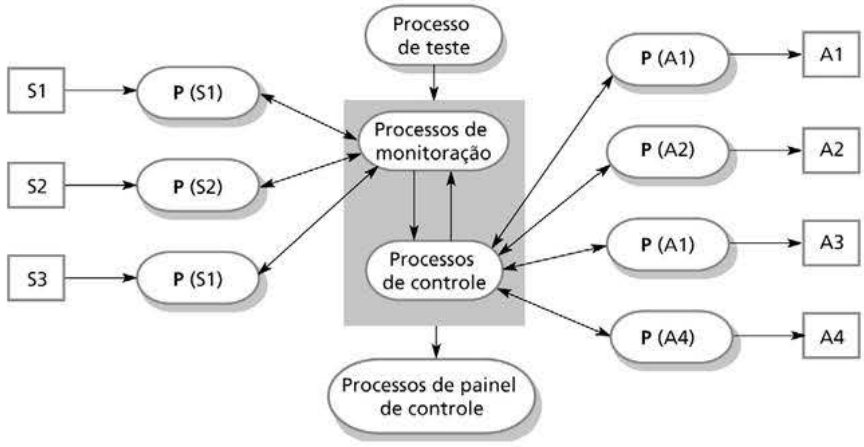


Tabela 15.1 Requisitos de tempo de estímulo–resposta

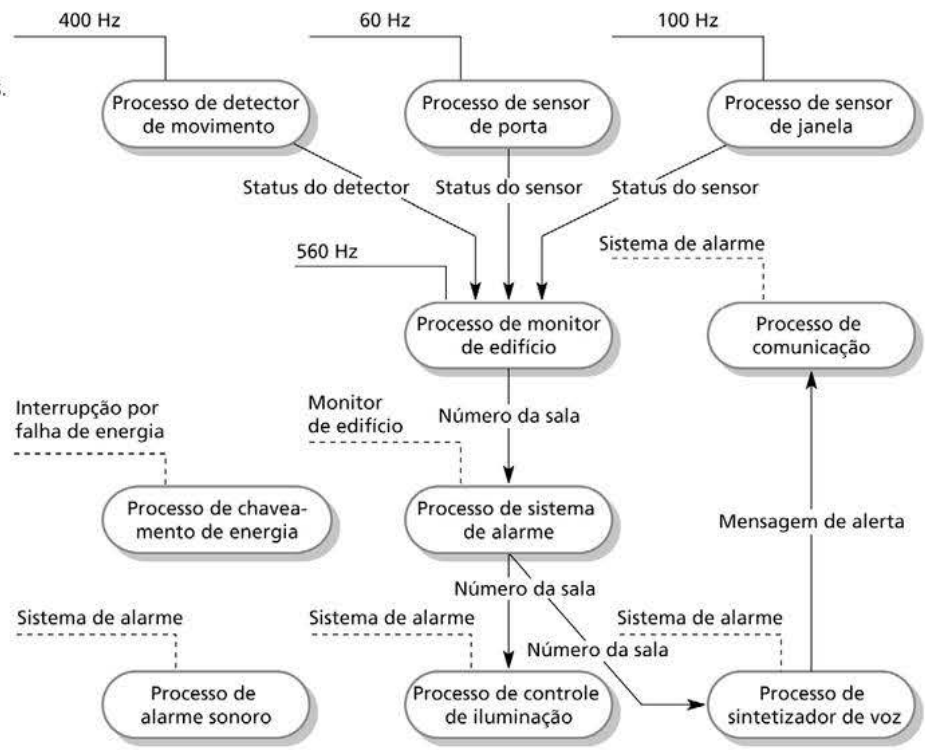
Estímulo–resposta	Requisitos de tempo
Interrupção por falha de energia	O chaveamento para a energia de reserva deve ser concluído dentro de um prazo de 50 ms.
Alarme de porta	Cada alarme de porta deve ser lido duas vezes por segundo.
Alarme de janela	Cada alarme de janela deve ser lido duas vezes por segundo.
Detector de movimento	Cada detector de movimento deve ser lido duas vezes por segundo.
Alarme sonoro	Um alarme sonoro deve ser acionado dentro de meio segundo após um alarme ter sido acionado por um sensor.
Interruptor de luzes	As luzes devem ser ligadas dentro de meio segundo após um alarme ter sido acionado por um sensor.
Comunicações	O chamado para a polícia deve ser iniciado dentro de 2 segundos após um alarme ter sido exposto por um sensor.
Sintetizador de voz	Uma mensagem sintetizada deve estar disponível dentro de 4 segundos após um alarme ter sido acionado por um sensor.

A alocação das funções de sistema para processos concorrentes é o próximo estágio de projeto. Existem três tipos de sensores que devem ser varridos periodicamente, cada um com um processo associado. Existe um sistema dirigido a interrupções para tratar a falha e o chaveamento de energia, um sistema de comunicações, um sintetizador de voz, um sistema de alarme sonoro e um sistema de acionamento de iluminação para ligar as luzes ao redor do sensor. Um processo independente controla cada um desses sistemas. Isso resulta na arquitetura de sistema mostrada na Figura 15.7.

Na Figura 15.7, as setas rotuladas ligam os processos, indicando os fluxos de dados entre eles com o rótulo que indica o tipo de fluxo de dados. Nem todos os processos recebem dados de outros processos. Por exemplo, o processo responsável pelo gerenciamento de uma falha de energia não requer dados oriundos de qualquer outro lugar do sistema.

A linha associada a cada processo na parte superior à esquerda é usada para indicar como o processo é controlado. As linhas sobre um processo periódico são linhas sólidas com o número mínimo de vezes que um processo deve ser executado

Figura 15.7
Arquitetura do processo de sistema de alarme de intrusos.



por segundo, conforme o rótulo. Os processos aperiódicos têm linhas tracejadas sobre seu canto superior esquerdo, que são rotuladas com o evento que causa a programação do processo.

O número de sensores a serem varridos e os requisitos de timing do sistema são usados para calcular em que freqüência cada processo deve ser programado. Por exemplo, existem 30 sensores de porta que devem ser verificados duas vezes por segundo. Isso significa que o processo de sensor de porta associado deve operar 60 vezes por segundo (60 Hz). O processo de detector de movimento deve operar 400 vezes por segundo, pois pode haver até 200 sensores de movimento no sistema. As informações de controle dos processos de atuador (por exemplo, controlador de alarme sonoro, controlador de iluminação etc.) indicam que eles são iniciados por um comando explícito, a partir do processo **Sistema de alarme** ou por uma **Interrupção por falta de energia**.

Esses processos podem ser implementados em Java por meio de threads. A Figura 15.8 mostra o código em Java que implementa o processo **BuildingMonitor**, que lê os sensores do sistema. Se esses sensores sinalizarem um intruso, o software

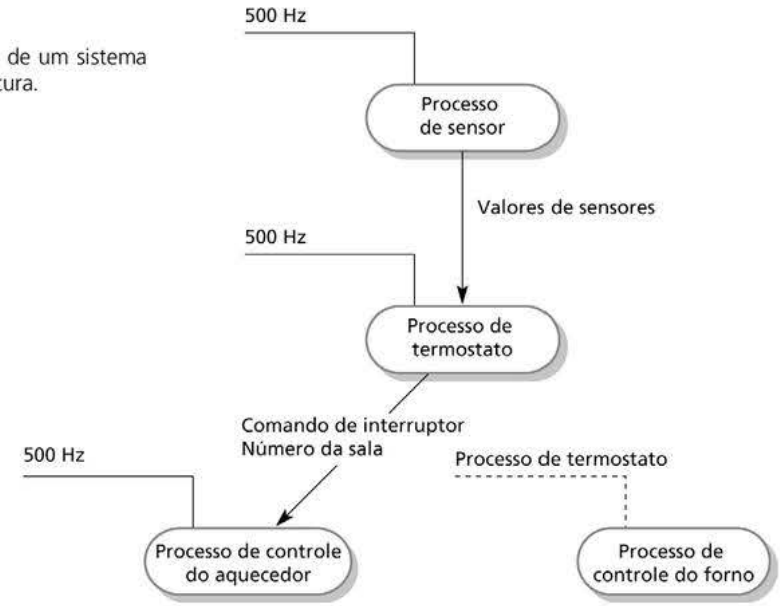
Figura 15.8

Implementação em Java do processo BuildingMonitor

```
// Veja http://www.software-engin.com/ para obter os links do código completo em Java deste
// exemplo
class BuildingMonitor extends Thread {
  BuildingSensor win, door, move ;
  Siren   siren = new Siren () ;
  Lights  lights = new Lights () ;
  Synthesizer synthesizer = new Synthesizer () ;
  DoorSensors doors = new DoorSensors (30) ;
  WindowSensors  windows = new WindowSensors (50) ;
  MovementSensors movements = new MovementSensors (200) ;
  PowerMonitor pm = new PowerMonitor () ;
  BuildingMonitor()
  {
     // ativar todos os sensores e iniciar os processos
     siren.start () ; lights.start () ;
     synthesizer.start () ; windows.start () ;
     doors.start () ; movements.start () ; pm.start () ;
  {}
  {public void run ()
  {{
     int room = 0 ;
     while (true)
     {
        // ler os sensores de movimento pelo menos duas vezes por segundo (400 Hz)
        move = movements.getVal () ;
        // ler os sensores de janela pelo menos duas vezes por segundo (100 Hz)
        win = windows.getVal () ;
        // ler os sensores de porta pelo menos duas vezes por segundo (60 Hz)
        door = doors.getVal () ;
        if (move.sensorVal = 1 | door.sensorVal = 1 | win.sensorVal = 1)
           {
             // um sensor indicou um intruso
             if (move.sensorVal = 1)  room = move.room ;
             if (door.sensorVal = 1)  room = door.room ;
             if (win.sensorVal = 1 )  room = win.room ;
             lights.on (room) ; siren.on () ; synthesizer.on (room) ;
             break ;
           }
     }
     lights.shutdown () ; siren.shutdown () ; synthesizer.shutdown () ;
     windows.shutdown () ; doors.shutdown () ; movements.shutdown () ;
  } // run
} //BuildingMonitor
```

Figura 15.9

Arquitetura do processo de um sistema de controle de temperatura.



ativará o sistema de alarme associado. Uso a linguagem Java padrão nesse código e considero que os requisitos de timing (incluídos como comentários) podem ser atendidos. Conforme expliquei anteriormente, a linguagem Java padrão não inclui recursos que permitem especificar a freqüência de execução de threads.

Após a definição da arquitetura do processo do sistema, os algoritmos para processamento de estímulos e geração de resposta devem ser projetados. Conforme expliquei na Seção 15.1, esse estágio de projeto detalhado é necessário no início do processo de projeto para assegurar que o sistema possa atender às suas restrições de timing especificadas. Se os algoritmos associados forem complexos, podem ser necessárias mudanças nos requisitos de timing. Contudo, a menos que o processamento de sinal seja necessário, os algoritmos de sistemas de tempo real são, muitas vezes, bem simples. Eles podem requerer, somente, a verificação de uma posição da memória, a realização de alguns processamentos simples ou o envio de um sinal. Conforme você pôde observar, na Figura 15.8 o processamento necessário em um sistema de alarme contra roubos segue esse modelo simples.

A etapa final do processo de projeto é projetar um sistema de programação que assegure que um processo será sempre programado para atender a seu prazo. Nesse exemplo, os prazos não são rígidos. As prioridades devem ser organizadas de modo que todos os processos de leitura de sensores tenham a mesma prioridade. O processo para tratamento de falha de energia deve ser um processo de nível de prioridade de interrupção mais alto. As prioridades dos processos que gerenciam o sistema de alarme devem ser as mesmas dos processos de sensor.

O sistema de alarme contra roubos é um sistema de monitoração, e não um sistema de controle, já que não inclui atuadores diretamente afetados pelos valores de sensores. Um exemplo de um sistema de controle seria o de aquecimento de um edifício. Esse sistema monitora os sensores de temperatura em diferentes salas do edifício e liga ou desliga a unidade de aquecedor dependendo da temperatura real e da temperatura definida no termostato da sala. O termostato controla também o chaveamento do forno do sistema.

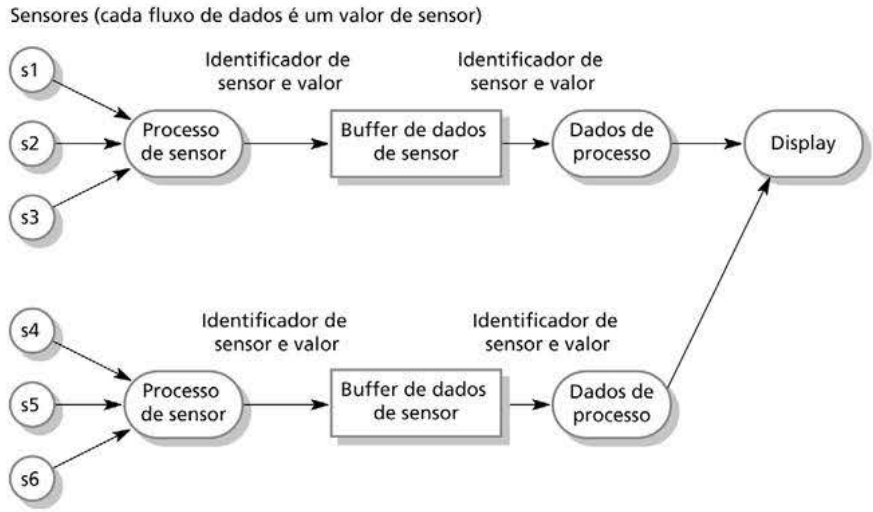
A arquitetura de processo desse sistema é mostrada na Figura 15.9. Está claro que sua forma geral é similar à do sistema de alarme contra roubos. Deixo para você desenvolver o projeto desse sistema mais detalhadamente.

15.4 Sistemas de aquisição de dados

Os sistemas de aquisição de dados coletam dados dos sensores para processamento e análise posterior. Esses sistemas são usados em circunstâncias em que os sensores estão coletando grandes quantidades de dados do ambiente do sistema e não é possível ou necessário processar os dados coletados em tempo real. Os sistemas de aquisição de dados são usados comumente em experimentos científicos e em sistemas de controle de processos nos quais os processos físicos, como uma reação química, ocorrem muito rapidamente.

Nos sistemas de aquisição de dados, os sensores podem gerar dados muito rapidamente, sendo que o problema principal é assegurar que uma leitura de sensor seja coletada antes que o valor do sensor mude. Isso leva a uma arquitetura genérica, conforme mostrada na Figura 15.10. A característica essencial da arquitetura de sistemas de aquisição de dados é que cada

Figura 15.10
Arquitetura genérica de sistemas de aquisição de dados.



grupo de sensores tem três processos associados. Esses processos são: o processo de sensor, que faz interface com o sensor e converte dados analógicos em valores digitais se necessário; o processo de buffer e o processo que consome os dados e realiza o processamento adicional.

Os sensores, naturalmente, podem ser de tipos diferentes, e o número de sensores em um grupo depende da taxa em que os dados chegam do ambiente. Na Figura 15.10, mostro dois grupos de sensores, **s1–s3** e **s4–s6**. Mostro também, à direita, um processo adicional que exibe os dados dos sensores. A maioria dos sistemas de aquisição de dados inclui processos de display e geração de relatórios que agregam os dados coletados e realizam o processamento adicional.

Como exemplo de um sistema de aquisição de dados, considere o modelo de sistema mostrado na Figura 15.11. Isso representa um sistema que coleta dados de sensores de monitoração de fluxo de nêutrons em um reator nuclear. Os dados de sensor são colocados em um buffer, do qual é extraído e processado, e o nível de fluxo médio é exibido no display do operador.

Cada sensor tem um processo associado que converte o nível de fluxo de entrada analógico (A) em um sinal digital (D). Ele passa esse nível de fluxo com o identificador de sensor para o buffer de dados de sensor. O processo responsável pelo processamento de dados retira os dados do buffer, executa o processamento e envia os dados a um processo de display para exibição em um console de operador.

Em sistemas de tempo real isso envolve a aquisição e o processamento de dados, e as velocidades e os períodos de execução do processo de aquisição (o produtor) e do processo de processamento (o consumidor) podem não estar sincronizados. Quando um processamento significativo é necessário, a aquisição de dados pode ser mais rápida do que o processamento de dados. Se somente cálculos simples precisarem ser realizados, o processamento poderá ser mais rápido do que a aquisição de dados.

Para amenizar essas diferenças de velocidade, os sistemas de aquisição de dados armazenam os dados de entrada usando um buffer em anel. O processo que produz os dados (o produtor) adiciona informações a esse buffer, e o processo que usa os dados (o consumidor) retira as informações do buffer (Figura 15.12).

Obviamente, a exclusão mútua deve ser implementada para evitar que os processos de produtor e de consumidor acessem o mesmo elemento no buffer ao mesmo tempo. O sistema deve também assegurar que o produtor não tente adicionar informações a um buffer cheio e o consumidor não retire informações de um buffer vazio.

Na Figura 15.13, é mostrada uma possível implementação do buffer de dados como um objeto Java. Os valores no buffer são do tipo **SensorRecord**, e existem duas operações definidas — chamadas **get** e **put**. A operação **get** retira um item do

Figura 15.11 Aquisição de dados de fluxo de nêutrons.

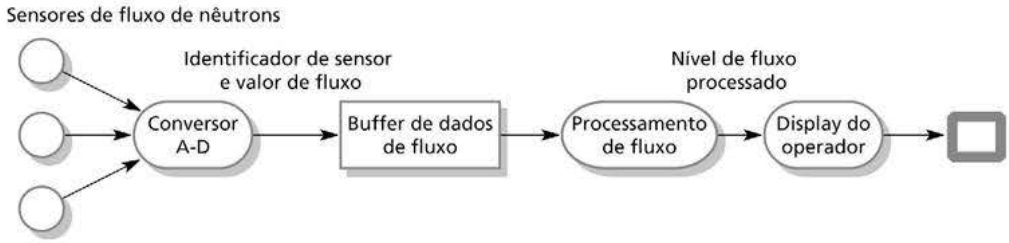


Figura 15.12

Um buffer em anel para aquisição de dados.

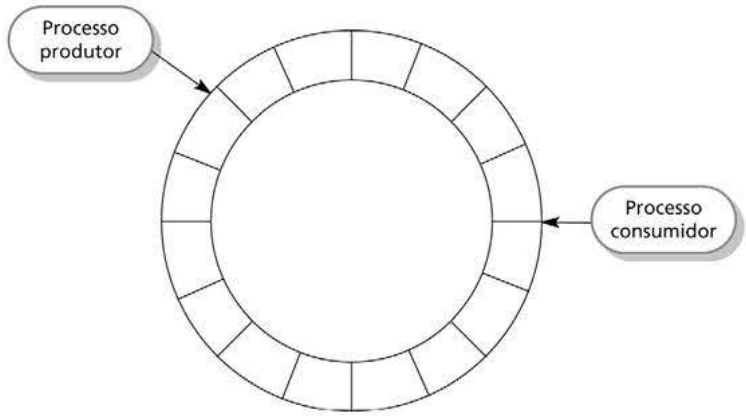


Figura 15.13

Uma implementação em Java de um buffer em anel.

```
class CircularBuffer
{
    int bufsize ;
    SensorRecord [] store ;
    int numberOfEntries = 0 ;
    int front = 0, back = 0 ;

    CircularBuffer (int n) {
        bufsize = n ;
        store = new SensorRecord [bufsize] ;
    } // CircularBuffer

    synchronized void put (SensorRecord rec ) throws InterruptedException
    {
        if ( numberOfEntries == bufsize)
            wait () ;
        store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
        back = back + 1 ;
        if (back == bufsize)
            back = 0 ;
        numberOfEntries = numberOfEntries + 1 ;
        notify () ;
    } // put

    synchronized SensorRecord get () throws InterruptedException
    {
        SensorRecord result = new SensorRecord (-1, -1) ;
        if (numberOfEntries == 0)
            wait () ;
        result = store [front] ;
        front = front + 1 ;
        if (front == bufsize)
            front = 0 ;
        numberOfEntries = numberOfEntries—1 ;
        notify () ;
        return result ;
    } // get
} // CircularBuffer
```

buffer e a operação **put** adiciona um item ao buffer. O construtor do buffer define seu tamanho quando os objetos do tipo **CircularBuffer** são declarados.

O modificador **synchronized** associado aos métodos **get** e **put** indica que esses métodos não devem operar concorrentemente. Quando um dos métodos é invocado, o sistema de tempo real obtém um bloqueio da instância de objeto para assegurar que o outro método não possa mudar o mesmo item no buffer. As invocações dos métodos **wait** e **notify** são usadas para assegurar que os itens não possam ser colocados em um buffer cheio ou retirados de um buffer vazio. O método **wait** causa a invocação do thread para se auto-suspender até que outro thread informe que ele deve parar de aguardar. Isso é feito pela chamada do método **notify**. Quando o método **wait** é chamado, o bloqueio do objeto protegido é liberado. O método **notify** ativa um dos threads que está aguardando e provoca o reinício de sua operação.

PONTOS-CHAVE

- Um sistema de tempo real é um sistema de software que deve responder a eventos em tempo real. Sua correção não depende somente dos resultados produzidos, mas também do momento em que esses resultados são produzidos.
- Um modelo geral para arquitetura de sistemas de tempo real envolve a associação de um processo a cada categoria de dispositivos de sensor e atuador. Outros processos de coordenação podem ser necessários.
- O projeto de arquitetura de um sistema de tempo real envolve, geralmente, a organização do sistema como um conjunto de processos concorrentes que interagem.
- Um sistema operacional de tempo real é responsável pelo gerenciamento de processos e recursos. Ele inclui sempre um programador, que é um componente responsável por decidir qual processo deve ser programado para execução. As decisões de programação são tomadas tendo em vista as prioridades dos processos.
- Os sistemas de monitoração e controle lêem periodicamente um conjunto de sensores que captam informações do ambiente do sistema. Eles tomam ações dependendo das leituras dos sensores, emitindo comandos para os atuadores.
- Os sistemas de aquisição de dados são, geralmente, organizados de acordo com um modelo produtor–consumidor. O processo produtor coloca os dados em um buffer em anel, onde são consumidos pelo processo consumidor. O buffer é implementado também como um processo e, assim, os conflitos entre produtor e consumidor são eliminados.

LEITURAS SUGERIDAS

Software engineering for real-time systems. Escrito sob uma perspectiva de engenharia em vez de ciência de computação, este livro é um bom guia prático para engenharia de sistemas de tempo real. Apresenta uma abrangência maior de questões de hardware do que o livro de Burns e Wellings e, portanto, é um excelente complemento para ele. (J. Cooling, 2003, Addison-Wesley.)

Real-time systems and programming languages, 3rd edition. Um texto excelente e abrangente que envolve todos os aspectos de sistemas de tempo real. (A. Burns e A. Wellings, 2001, Addison-Wesley.)

Doing hard time: developing real-time systems with UML, objects, frameworks and patterns. Este livro explica como as técnicas orientadas a objetos podem ser usadas no projeto de sistemas de tempo real. À medida que as velocidades de hardware aumentam, essa abordagem está se tornando cada vez mais viável para projeto de sistemas de tempo real. (B. P. Douglass, 1999, Addison-Wesley.)

EXERCÍCIOS

15.1 Explique, usando exemplos, por que os sistemas de tempo real devem, geralmente, ser desenvolvidos com a utilização de processos concorrentes.

15.2 Explique por que uma abordagem orientada a objetos para desenvolvimento de software pode não ser adequada para sistemas de tempo real.

15.3 Desenhe modelos de máquina de estado de software de controle para os seguintes sistemas:

- Uma máquina de lavar roupa automática com diferentes programas para diferentes tipos de roupas.
- Um software de um equipamento de reprodução de CDs.
- Um aparelho de secretária eletrônica que registra as mensagens recebidas e exibe o número de mensagens aceitas em um display de LEDs. O sistema deve permitir que o proprietário do telefone realize uma ligação, digite uma seqüência de números (identificados por tons) e tenha as mensagens registradas reproduzidas pelo telefone.
- Uma máquina de venda de café pode fornecer café com ou sem leite e açúcar. O usuário deposita uma moeda e faz sua escolha pressionando um botão da máquina. Isso provoca a saída de um copo com café em pó. O usuário coloca esse copo sob uma torneira, pressiona outro botão e a água quente é fornecida.

15.4 Usando as técnicas de projeto de sistema de tempo real explicadas neste capítulo, refaça o projeto do sistema de coleta de dados da estação meteorológica, abordado no Capítulo 14, como um sistema de estímulo–resposta.

15.5 Projete uma arquitetura de processo para um sistema de monitoração de ambiente que coleta dados de um conjunto de sensores de qualidade de ar situado em torno de uma cidade. Existem 5.000 sensores organizados em 100 regiões. Cada sensor deve ser interrogado quatro vezes por segundo. Quando mais de 30% dos sensores de determinada região indicarem que a qualidade do ar está abaixo de um nível aceitável, luzes locais de advertência serão ativadas. Todos os sensores retornam as leituras a um computador central que gera relatórios a cada 15 minutos sobre a qualidade de ar na cidade.

15.6 Explique os pontos fortes e fracos da linguagem Java como uma linguagem de programação para sistemas de tempo real. Em que extensão os problemas de programação em Java desaparecerão quando processadores mais rápidos forem usados?

15.7 Um sistema de proteção de trem aciona automaticamente os freios de um trem caso o limite de velocidade de um trecho da via seja excedido ou se o trem entrar em um trecho da via que esteja, no momento, sinalizado com uma luz vermelha (como, por exemplo, no caso de o segmento não dever ser usado). Os detalhes são mostrados no Quadro 15.1, abaixo. Identifique os estímulos que devem ser processados por um sistema de controle de bordo do trem e as respostas associadas aos estímulos.

15.8 Sugira uma possível arquitetura de processo para esse sistema. Documente a arquitetura de processo usando a notação mostrada na Figura 15.7, identificando claramente se os estímulos são periódicos ou aperiódicos.

15.9 Se um processo periódico no sistema de proteção de bordo do trem for usado para coletar dados de um transmissor da via, com que freqüência o processo deve ser programado para assegurar que o sistema, com certeza, colete as informações do transmissor? Explique como você chegou à sua resposta.

15.10 Foi solicitado que você trabalhe em um projeto de desenvolvimento de tempo real para uma aplicação militar, mas você não tem experiência em projetos desse domínio. Explique o que você, como engenheiro de software profissional, deve fazer antes de iniciar o trabalho no projeto.

Quadro 15.1 Descrição do sistema de proteção de trem

- O sistema adquire informações de limite de velocidade de um trecho, a partir de um transmissor da via, que transmite continuamente o identificador do trecho e seu limite de velocidade. O mesmo transmissor transmite também informações sobre o status do sinal que controla esse segmento da via. O tempo necessário para transmitir as informações de segmento da via e de sinal é de 50 milissegundos.
- O trem pode receber informações do transmissor da via quando estiver a 10 m de um transmissor.
- A velocidade máxima do trem é de 180 km/h.
- Os sensores do trem fornecem informações sobre a velocidade atual do trem (atualizada a cada 250 milissegundos) e o status dos freios do trem (atualizado a cada 100 milissegundos).
- Se a velocidade do trem exceder o limite de velocidade do trecho atual em mais de 5 km/h, um aviso será emitido na cabine do maquinista. Se a velocidade do trem exceder o limite de velocidade do segmento atual em mais de 10 km/h, os freios do trem serão, automaticamente, acionados até que a velocidade caia para o limite de velocidade do trecho. Os freios do trem devem ser acionados dentro de 100 milissegundos a partir do momento em que o excesso de velocidade do trem for detectado.
- Se o trem entrar em um trecho da via sinalizado com uma luz vermelha, o sistema de proteção do trem acionará os freios e reduzirá a velocidade para zero. Os freios do trem devem ser acionados dentro de 100 milissegundos a partir do momento em que o sinal de luz vermelha for recebido.
- O sistema atualiza, continuamente, um display de status na cabine do maquinista.

16

Projeto de interface com o usuário

Objetivos

O objetivo deste capítulo é apresentar alguns aspectos de projeto de interface com o usuário que são importantes para os engenheiros de software. Após ler este capítulo, você:

- compreenderá uma série de princípios de projeto de interface com o usuário;
- terá sido apresentado a vários estilos de interação e compreenderá quando esses estilos são mais apropriados;
- compreenderá quando usar a apresentação de informações gráficas e textuais;
- saberá o que está envolvido nas atividades principais do processo de projeto de interface com o usuário;
- compreenderá os atributos de usabilidade e terá sido apresentado às diferentes abordagens para avaliação de interfaces.

Conteúdo

16.1 Tópicos de projeto
16.2 Processo de projeto de UI
16.3 Análise de usuário
16.4 Prototipação de interface com o usuário
16.5 Avaliação de interface

O projeto de sistema de computador abrange um espectro de atividades desde o projeto de hardware até o projeto de interface com o usuário. Enquanto os especialistas são, com freqüência, contratados para o projeto de hardware e para projetos gráficos de páginas Web, somente grandes organizações normalmente contratam projetistas especializados em interface para seu software de aplicação. Portanto, os engenheiros de software devem, muitas vezes, assumir a responsabilidade pelo projeto de interface com o usuário, bem como pelo projeto do software para implementar essa interface.

Mesmo quando os projetistas e programadores de software são usuários competentes de tecnologias de implementação de interfaces, como classes Swing do Java (Elliot, et al., 2002) ou XHTML (Musciano e Kennedy, 2002), as interfaces com o usuário que desenvolvem não são, freqüentemente, atraentes e apropriadas para seus usuários-alvo. Portanto, enfoco o processo de

projeto para interfaces com o usuário em vez do software que implementa esses recursos. Devido às limitações de espaço, considero somente interfaces gráficas com o usuário. Não explico interfaces que requerem displays especiais (talvez mais simples), como telefones celulares, aparelhos de DVD, televisões, máquinas copiadoras e de fax. Naturalmente, posso somente apresentar o tópico neste capítulo e recomendo textos, como os de Dix et al. (Dix, et al., 2004), Weiss (Weiss, 2002) e Shneiderman (Shneiderman, 1998) para obter mais informações sobre projeto de interface com o usuário.

O projeto cuidadoso de interface com o usuário é uma parte essencial de todo o processo de projeto de software. Se um sistema de software deve atingir todo o seu potencial, é essencial que sua interface com o usuário seja projetada para combinar as habilidades, experiências e expectativas dos usuários previstos. Um bom projeto de interface com o usuário é crítico para a confiabilidade do sistema. Muitos dos chamados 'erros de usuário' são causados pelo fato de que as interfaces de usuário não consideram as capacidades dos usuários reais e seu ambiente de trabalho. Uma interface com o usuário projetada inadequadamente significa que os usuários serão, provavelmente, incapazes de acessar algumas das características do sistema, cometerão erros e sentirão que o sistema os limitará em vez de os ajudar a atingir o objetivo para o qual o sistema é usado.

Ao tomar decisões de projeto de interface com o usuário, você deve levar em conta as capacidades físicas e mentais das pessoas que usam o software. Não cabe neste capítulo explicar questões humanas detalhadamente, mas os fatores importantes a serem considerados são:

1. As pessoas têm uma memória limitada de curto prazo — não podemos lembrar instantaneamente de sete itens de informação (Miller, 1957). Portanto, se você apresentar aos usuários uma quantidade muito elevada de informações ao mesmo tempo, eles podem não ser capazes de assimilá-las.
2. Todos nós cometemos erros, especialmente quando temos de lidar com uma quantidade muito grande de informações ou quando estamos sob estresse. Quando os sistemas não funcionam direito e emitem mensagens e avisos, tal fato impõe, muitas vezes, mais estresse sobre os usuários, aumentando, portanto, as chances de que eles cometam erros operacionais.
3. Temos uma faixa variada de capacidades físicas. Algumas pessoas vêem e ouvem melhor do que outras, algumas pessoas são daltônicas e algumas são melhores do que outras em manipulação física. Você não deve projetar para suas próprias capacidades e presumir que todos os outros usuários serão capazes de lidar com isso.
4. Temos preferências diferentes de interação. Algumas pessoas gostam de trabalhar com figuras, outras com texto. A manipulação direta é natural para algumas pessoas, mas outras preferem um estilo de interação baseado em emissão de comandos para o sistema.

Esses fatores humanos são a base dos princípios de projeto, mostrados na Tabela 16.1. Os princípios gerais são aplicáveis a todos os projetos de interface com o usuário e devem ser, normalmente, instanciados como diretrizes de projeto mais detalhadas para organizações ou tipos de sistema específicos. Os princípios de projeto de interface com o usuário são abordados mais detalhadamente por Dix (Dix, et al., 2004). Shneiderman (Shneiderman, 1998) fornece uma lista maior de diretrizes de projeto mais específicas de interface com o usuário.

O princípio da *familiaridade do usuário* sugere que os usuários não devem ser forçados a se adaptar a uma interface porque a implementação desta é conveniente. A interface deve usar termos familiares ao usuário, e os objetos manipulados pelo sistema devem estar diretamente relacionados ao ambiente de trabalho do usuário. Por exemplo, se um sistema é projetado

Tabela 16.1 Princípios de projeto de interface de usuário

Princípio	Descrição
Familiaridade de usuário	A interface deve usar termos e conceitos obtidos da experiência de pessoas que farão mais uso do sistema.
Consistência	A interface deve ser consistente de maneira que, sempre que possível, as operações comparáveis sejam ativadas da mesma maneira.
Surpresa mínima	Os usuários nunca devem ser surpreendidos pelo comportamento de um sistema.
Facilidade de recuperação	A interface deve incluir mecanismos que permitam aos usuários se recuperarem de erros.
Guia de usuário	A interface deve fornecer feedback significativo quando ocorrerem erros e fornecer recursos sensíveis ao contexto para ajudar o usuário.
Diversidade de usuário	A interface deve fornecer recursos de interação adequados para tipos diferentes de usuários de sistema.

para uso pelos controladores de tráfego aéreo, os objetos manipulados devem ser aeronaves, rotas de vôos, sinalizações etc. As operações associadas podem aumentar ou reduzir a velocidade da aeronave, ajustar o 'teto' e mudar a altura. A implementação da base da interface, em termos de arquivos e estruturas de dados, não deve ser revelada ao usuário final.

O princípio da *consistência de interface com o usuário* significa que os comandos e menus de sistema devem ter o mesmo formato, os parâmetros devem ser passados para todos os comandos da mesma maneira e o acionamento dos comandos deve ser similar. As interfaces consistentes reduzem o tempo de aprendizado do usuário. O conhecimento obtido em um comando ou aplicação é, portanto, aplicável a outras partes do sistema ou a aplicações relacionadas.

A consistência da interface ao longo das aplicações é também importante. Tanto quanto possível, os comandos com significados similares em aplicações diferentes devem ser expressos da mesma maneira. Os erros são freqüentemente cometidos quando o mesmo comando de teclado, como 'Ctrl+B', significam coisas diferentes em sistemas diferentes. Por exemplo, no processador de texto que uso normalmente, 'Ctrl+B' significa texto em negrito, mas no programa de gráficos que uso para desenhar diagramas, 'Ctrl+B' significa mover o objeto selecionado para trás de um outro objeto. Cometo erros quando uso os programas juntos e, algumas vezes, tento colocar um texto em negrito em um diagrama usando a combinação de teclas. Depois, fico confuso quando o texto desaparece atrás do objeto marcado. Você pode evitar esse tipo de erro se seguir os atalhos de teclas de comando definidos pelo sistema operacional usado.

Esse nível de consistência é de nível baixo. Os projetistas de interface devem sempre tentar atingir esse nível em uma interface com o usuário. Uma consistência de nível alto é também, algumas vezes, desejável. Por exemplo, pode ser apropriado apoiar as mesmas operações (imprimir, copiar etc.) em todos os tipos de entidades de sistema. Contudo, Grudin (Grudin, 1989) observa que a consistência completa não é possível nem desejável. Pode ser sensato implementar uma operação de exclusão em um computador arrastando entidades para dentro de uma lixeira. Seria incômodo excluir texto em um processador de texto dessa maneira.

Infelizmente, os princípios da familiaridade do usuário e da consistência do usuário são, algumas vezes, conflitantes. De modo ideal, as aplicações com características comuns devem sempre usar os mesmos comandos para acessar essas características. Contudo, isso pode conflitar com a prática do usuário quando os sistemas são projetados para apoiar determinado tipo de usuário, como os projetistas gráficos. Esses usuários podem ter desenvolvido seu próprio estilo de interações, terminologia e convenções operacionais. Os estilos podem estar em conflito com 'padrões' de interação apropriados a aplicações mais gerais, como processadores de texto.

O princípio da *surpresa mínima* é apropriado, pois as pessoas ficam muito irritadas quando um sistema se comporta de maneira inesperada. Quando um sistema é usado, os usuários constroem um modelo mental de como o sistema funciona. Se uma ação em um contexto causa determinado tipo de mudança, é razoável esperar que a mesma ação em um contexto diferente causará uma mudança semelhante. Se algo completamente diferente ocorrer, o usuário ficará surpreso e confuso. Os projetistas de interface devem, portanto, tentar assegurar que as ações semelhantes tenham efeitos semelhantes.

As surpresas nas interfaces com o usuário são, muitas vezes, o resultado do fato de que muitas interfaces são baseadas em modos. Isso significa que existem diversos modos de trabalhar (por exemplo, modo de visualizar e modo de editar), e o efeito de um comando é diferente dependendo do modo. É muito importante que, ao projetar uma interface, você inclua um indicador visual que mostre ao usuário o modo de trabalho atual.

O princípio da *facilidade de recuperação* é importante, pois os usuários inevitavelmente cometem erros quando estão usando um sistema. O projeto de interface pode minimizar esses erros (por exemplo, o uso de menus evita erros de digitação), mas os erros nunca poderão ser completamente eliminados. Conseqüentemente, você deve incluir recursos de interface que permitam aos usuários se recuperarem de seus erros. Esses recursos podem ser de três tipos:

1. *Confirmação de ações destrutivas.* Se um usuário especifica uma ação que é potencialmente destrutiva, o sistema deve solicitar ao usuário uma confirmação de se isso é realmente o que ele deseja antes de destruir qualquer informação.
2. *Disponibilidade de um recurso do tipo 'refazer'.* O 'refazer' restaura o sistema para um estado anterior ao da ação ocorrida. Diversos níveis de 'refazer' são úteis, pois os usuários nem sempre reconhecem imediatamente que um erro foi cometido.
3. *Checkpointing.* O checkpointing envolve a gravação do estado do sistema em intervalos periódicos e a permissão de o sistema ser reiniciado a partir do último checkpointing. Assim, quando ocorrem erros, os usuários podem retornar a um estado anterior e começar novamente. Muitos sistemas, atualmente, incluem checkpointing para lidar com falhas de sistema, mas, paradoxalmente, não permitem que os usuários do sistema usem o recurso para recuperar-se de seus próprios erros.

Um princípio relacionado é o de *guia do usuário.* As interfaces devem ter recursos built-in de assistência ou ajuda ao usuário. Esses recursos devem ser integrados ao sistema e fornecer níveis diferentes de ajuda e orientação. Os níveis devem variar desde informações básicas sobre iniciar o sistema até uma descrição completa dos recursos deste.

Os recursos de ajuda devem ser estruturados e, desse modo, os usuários não são sobrecarregados com informações ao solicitarem ajuda.

O princípio da *diversidade do usuário* reconhece que, para muitos sistemas interativos, pode haver tipos diferentes de usuário. Alguns serão usuários casuais que interagem ocasionalmente com o sistema, enquanto outros podem ser usuários experientes que usam o sistema várias horas por dia. Os usuários casuais necessitam de interfaces que forneçam um guia, mas os usuários experientes exigem atalhos, de modo que possam interagir o mais rapidamente possível. Além disso, os usuários podem sofrer de problemas físicos de vários tipos e, se possível, a interface deve ser adaptável a isso. Portanto, você pode incluir recursos para exibir o texto ampliado, substituir som por texto, produzir botões grandes etc. Isso reflete a noção de Projeto Universal (UD — *Universal Design*) (Preiser e Ostoff, 2001), filosofia de projeto cuja meta é evitar a exclusão de usuários devido a opções descuidadas de projeto.

O princípio de reconhecimento da diversidade do usuário pode conflitar com os outros princípios de projeto de interface, já que alguns usuários podem preferir interação muito rápida a, por exemplo, consistência da interface com o usuário. Similarmente, o nível de guia exigido pode ser radicalmente diferente para usuários diferentes, e pode ser impossível desenvolver um apoio adequado a todos os tipos de usuários. Portanto, você deve achar um ponto de conciliação que satisfaça as necessidades desses usuários.

16.1 Tópicos de projeto

Antes de continuar a explicar o processo de projeto de interface com o usuário, explico alguns tópicos gerais de projeto a serem considerados pelos projetistas de interface com o usuário (UI — *User Interface*). Essencialmente, o projetista de uma interface com o usuário para computador enfrenta duas questões principais:

1. Como o usuário deve interagir com o sistema de computador?
2. Como as informações de um computador devem ser apresentadas ao usuário?

Uma interface com o usuário coerente deve integrar a interação do usuário e a apresentação de informações. Isso pode ser difícil, pois o projetista precisa encontrar uma conciliação entre os estilos de interação e apresentação mais apropriados para a aplicação, os conhecimentos e as experiências dos usuários do sistema, e o equipamento disponível.

16.1.1 Interação do usuário

A interação do usuário significa emitir comandos e dados associados ao sistema de computador. Nos primeiros computadores, a única forma de fazer isso era por meio de uma interface de linha de comando, e uma linguagem de propósito especial era usada para se comunicar com a máquina. Entretanto, isso era utilizado por usuários especialistas e uma série de abordagens desenvolvidas são atualmente mais fáceis de usar. Shneiderman (Shneiderman, 1998) classificou essas formas de interação em cinco estilos primários:

1. *Manipulação direta.* O usuário interage diretamente com os objetos na tela. A manipulação direta envolve, geralmente, um dispositivo apontador (um mouse, um stylus, um trackball ou o próprio dedo sobre telas sensíveis a toque), que indica o objeto a ser manipulado, e a ação, que especifica o que deve ser feito com esse objeto. Por exemplo, para excluir um arquivo, você pode clicar sobre um ícone que representa esse arquivo e arrastá-lo para um ícone de lixeira.
2. *Seleção de menu.* O usuário seleciona um comando de uma lista de possibilidades (um menu). O usuário pode também selecionar um outro objeto da tela por manipulação direta e o comando opera sobre esse objeto. Nessa abordagem, para excluir um arquivo, você deve selecionar o ícone de arquivo e selecionar o comando de exclusão.
3. *Preenchimento de formulários.* O usuário preenche os campos de um formulário. Alguns campos podem ter menus associados e o formulário pode ter 'botões' de ação que, quando pressionados, causam o início de alguma ação. Não é aconselhável usar normalmente essa abordagem para implementar a interface de operações, como excluir um arquivo. Se você fizer dessa forma, isso envolverá o preenchimento do nome do arquivo no formulário e, depois, pressionar um botão para excluir.
4. *Linguagem de comando.* O usuário emite um comando especial e parâmetros associados para instruir o sistema sobre o que fazer. Para excluir um arquivo, você deve digitar um comando para excluir com o nome do arquivo como parâmetro.
5. *Linguagem natural.* O usuário emite um comando em linguagem natural. Esse geralmente é o estágio inicial de uma linguagem de comando; a linguagem natural é analisada e traduzida em comandos de sistema. Para excluir um arquivo, você pode digitar 'excluir o arquivo denominado xxx'.

Cada um desses estilos de interação tem vantagens e desvantagens e é mais adequado a determinado tipo de aplicação e usuário (Shneiderman, 1998). A Tabela 16.2 mostra as principais vantagens e desvantagens desses estilos e sugere tipos de aplicações em que eles poderiam ser usados.

Naturalmente, os estilos de interação podem ser combinados com muitos outros estilos usados na mesma aplicação. Por exemplo, o Windows da Microsoft apóia a manipulação direta dos ícones de representação de arquivos e diretórios, seleção de comando baseado em menus, e, para comandos como os de configuração, o usuário deve preencher um formulário de propósito especial apresentado.

Em princípio, deve ser possível separar o estilo de interação das entidades de base manipuladas por meio da interface com o usuário. Essa foi a base do modelo Seeheim (Pfaff e ten Hagen, 1985) de gerenciamento de interface com o usuário. Nesse modelo, a apresentação de informações, o gerenciamento de diálogos e a aplicação são separados. Na realidade, esse modelo é mais ideal do que prático, mas é certamente possível ter interfaces separadas para diferentes categorias de usuários (usuários casuais ou usuários experientes, por exemplo) que interagem com a mesma base de sistema. Isso é ilustrado na Figura 16.1, que mostra uma interface de linguagem de comando e uma interface gráfica para um sistema operacional de base, como o Linux.

As interfaces com o usuário baseadas na Web têm como ponto de partida o apoio fornecido pelo HTML ou XHTML (as linguagens de descrição de páginas usadas para páginas Web) junto com linguagens como Java, que podem associar programas a componentes de uma página. Por causa de essas interfaces baseadas na Web serem projetadas geralmente para usuários casuais, eles usam, na maioria das vezes, interfaces baseadas em formulários. É possível construir interfaces de manipulação direta na Web, mas essa é uma tarefa de programação complexa. Além disso, devido à faixa de experiência de usuários da Web e o fato de que eles vêm de culturas diferentes, é difícil estabelecer uma metáfora de interface com o usuário para interação direta que seja aceitável universalmente.

Para ilustrar o projeto de interação com o usuário baseada em Web, explico a abordagem usada no sistema LIBSYS na qual os usuários podem acessar documentos de outras bibliotecas. Existem duas operações fundamentais que precisam ser apoiadas:

1. *Busca de documentos* na qual os usuários usam os recursos de busca para encontrar os documentos de que necessitam.
2. *Solicitação de documentos* na qual os usuários escolhem se o documento deve ser entregue em sua máquina local ou no servidor para impressão.

A interface com o usuário do LIBSYS é implementada por meio de um navegador Web e, assim, considerando que os usuários devem fornecer informações ao sistema, como o identificador do documento, seu nome e seus detalhes de autorização, faz sentido usar uma interface baseada em formulários. A Figura 16.2 mostra um possível projeto de interface para o componente de busca do sistema.

Tabela 16.2 Vantagens e desvantagens de estilos de interação

Estilo de interação	Principais vantagens	Principais desvantagens	Exemplos de aplicação
Manipulação direta	Interação rápida e intuitiva Facilidade de aprendizado	Pode ser difícil implementar É adequado somente quando houver uma metáfora visual para tarefas e objetos	Jogos de vídeo Sistemas CAD
Seleção de menu	Evita erros do usuário É necessário pouca digitação	Lento para usuários experientes Pode tornar-se complexo se houver muitas opções de menus	A maioria dos sistemas de propósito geral
Preenchimento de formulários Facilidade de aprendizado verificável	Entrada de dados simples	Ocupa grande quantidade de espaço de tela Causa problemas quando as opções de usuário não combinam com os campos de formulário	Controle de estoque Processamento de empréstimos pessoais
Linguagem de comando	Poderosa e flexível	Dificuldade de aprendizado Gerenciamento inadequado de erros	Sistemas operacionais Sistemas de comando e controle
Linguagem natural	Acessível a usuários casuais Facilmente estendido	Requer muita digitação Não são confiáveis	Sistemas de recuperação de informações

Figura 16.1
Interfaces com vários usuários.

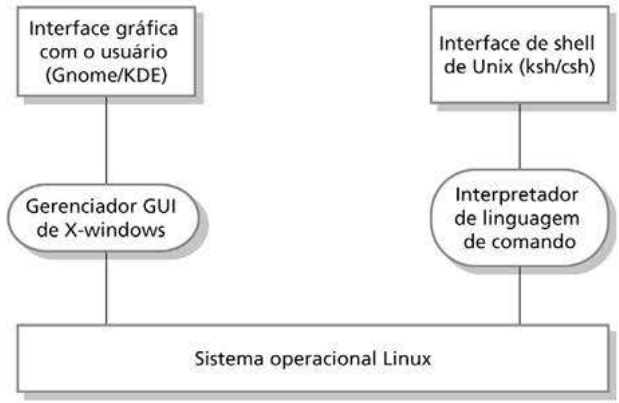
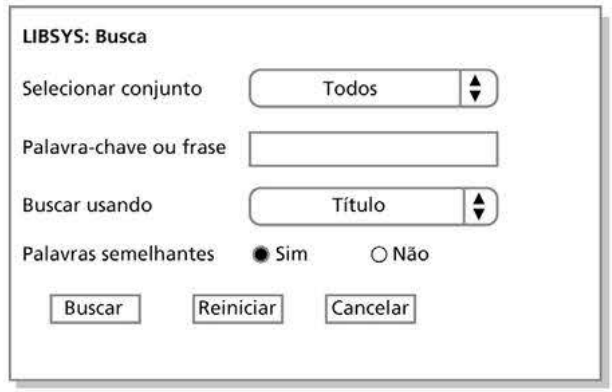


Figura 16.2
Interface baseada em formulários para o sistema LIBSYS.



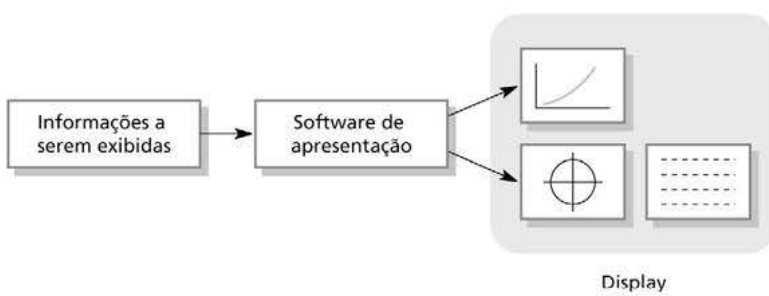
Nas interfaces baseadas em formulários, o usuário fornece todas as informações necessárias e, em seguida, inicia a ação pressionando um botão. Os campos de formulários podem ser menus, campos de entrada de texto livre ou botões de seleção. No exemplo do LIBSYS, um usuário escolhe um conjunto a ser pesquisado com base em um menu de conjuntos que podem ser acessados ('Todos' é o padrão, que significa todos os conjuntos) e os tipos de frase de busca em um campo de entrada de texto livre. O usuário escolhe o campo de registro de biblioteca baseado em um menu ('Título' é o padrão) e marca um botão de seleção para indicar se devem ser pesquisados termos de busca semelhantes ao registro.

16.1.2 Apresentação de informações

Todos os sistemas interativos têm de fornecer algum modo de apresentação de informações aos usuários. A apresentação de informações pode ser simplesmente uma representação direta das informações de entrada (por exemplo, texto em um processador de texto) ou pode apresentar as informações graficamente. Uma boa diretriz de projeto é manter o software necessário para apresentação de informações separado das informações em si. A separação do sistema de apresentação dos dados permite alterar a representação na tela do usuário sem ter de alterar a base do sistema computacional. Isso é ilustrado na Figura 16.3.

A abordagem MVC (Figura 16.4), inicialmente disponível em Smalltalk (Golberg e Robson, 1983), é uma maneira eficiente de apoiar as várias apresentações de dados. Os usuários podem interagir com cada apresentação em um estilo

Figura 16.3
Apresentação de informações.



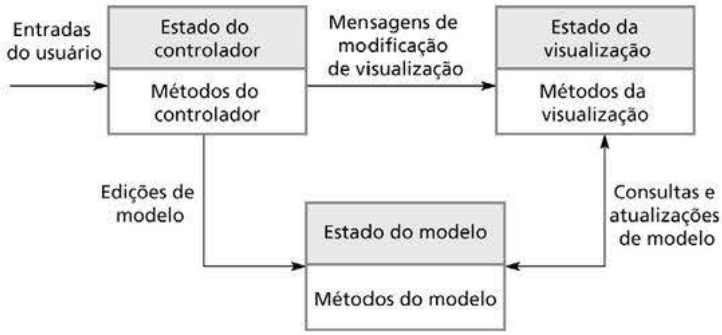


Figura 16.4
Modelo MVC de interação com o usuário.

apropriado à apresentação. Os dados a serem exibidos são encapsulados por um objeto de modelo. Cada objeto de modelo pode ter uma série de objetos de visualização separados, associados a ele, sendo cada visualização uma apresentação de display diferente do modelo.

Cada visualização tem um objeto controlador associado que cuida das entradas de usuário e da interação de dispositivo. Portanto, um modelo que representa dados numéricos pode ter uma visualização que represente os dados como um histograma e uma visualização que apresente os dados como uma tabela. O modelo pode ser editado pela mudança de valores na tabela ou pelo aumento ou diminuição das barras no histograma. Explico isso detalhadamente no Capítulo 18, no qual mostro como você pode usar o padrão Observer para implementar o framework de MVC.

Para encontrar a melhor apresentação de informações, você precisa conhecer as experiências dos usuários e como eles usam o sistema. Ao decidir como apresentar as informações, devem ser consideradas as seguintes questões:

1. O usuário está interessado em informações precisas ou nos relacionamentos entre os valores de dados?
2. Com que rapidez mudam os valores de informações? A mudança de um valor deve ser indicada imediatamente para o usuário?
3. O usuário deve executar alguma ação em resposta a uma mudança na informação?
4. O usuário necessita interagir com as informações exibidas por meio de uma interface de manipulação direta?
5. As informações a serem exibidas são textuais ou numéricas? Os valores relativos dos itens de informação são importantes?

Você não deve presumir que o uso de imagens torne seu display mais interessante. As imagens ocupam espaços de tela valiosos (uma questão importante em dispositivos portáteis) e podem levar um longo tempo para carregar um arquivo, caso o usuário esteja trabalhando sobre uma conexão lenta e discada.

As informações que não mudam durante uma sessão podem ser apresentadas tanto graficamente quanto como texto, dependendo da aplicação. A apresentação textual ocupa menos espaço na tela, mas não pode ser lida de relance. Você deve distinguir informações que não mudam de informações dinâmicas pelo uso de um estilo de apresentação diferente. Por exemplo, você pode apresentar todas as informações estáticas em determinada fonte ou cor, ou pode associar um ícone de 'informação estática' a elas.

Você deve usar texto quando forem necessárias informações precisas e estas mudarem com relativa lentidão. Se os dados mudarem rapidamente ou se os relacionamentos entre os dados forem significativos, mais do que seus valores precisos, então você deve apresentar as informações graficamente.

Por exemplo, considere um sistema que registra e resume os valores de vendas mensais de uma empresa. A Figura 16.5 ilustra como as mesmas informações podem ser apresentadas em forma de texto ou graficamente. Os gerentes que estudam os valores de vendas estão, geralmente, mais interessados nas tendências ou números anômalos em vez de em valores precisos. A apresentação gráfica dessas informações como um histograma destaca os números anômalos, em março e maio, dos outros. A Figura 16.5 ilustra também como a apresentação textual ocupa menos espaço do que uma representação gráfica das mesmas informações.

Em salas de controle ou painéis de instrumentos, como o de um automóvel, as informações apresentadas representam o estado de algum outro sistema (por exemplo, a altitude de uma aeronave) e estão mudando a todo momento. Um display digital que muda constantemente pode ser confuso e irritante quando os leitores não podem ler ou assimilar as informações antes da mudança. Essas informações numéricas que variam dinamicamente são, portanto, mais bem apresentadas graficamente com o uso de uma representação analógica. O display gráfico pode ser complementado, se necessário, com um display digital preciso. As diferentes maneiras de apresentação de informações numéricas dinâmicas são mostradas na Figura 16.6.

Os displays analógicos contínuos dão ao observador certo senso de valor relativo. Na Figura 16.7, os valores de temperatura e pressão são aproximadamente os mesmos. Contudo, o display gráfico mostra que a temperatura está próxima de

Figura 16.5

Apresentações alternativas de informações.

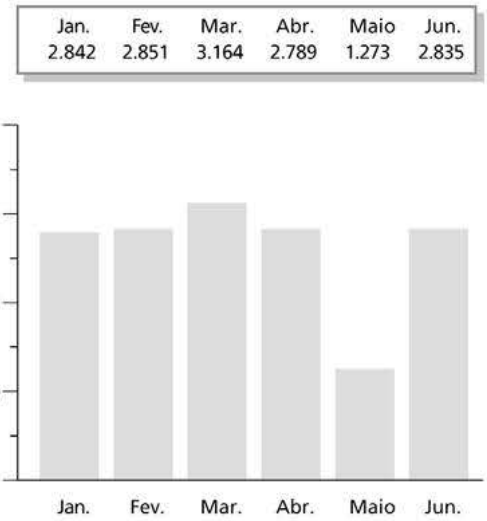


Figura 16.6

Métodos de apresentação de informações numéricas que variam dinamicamente.

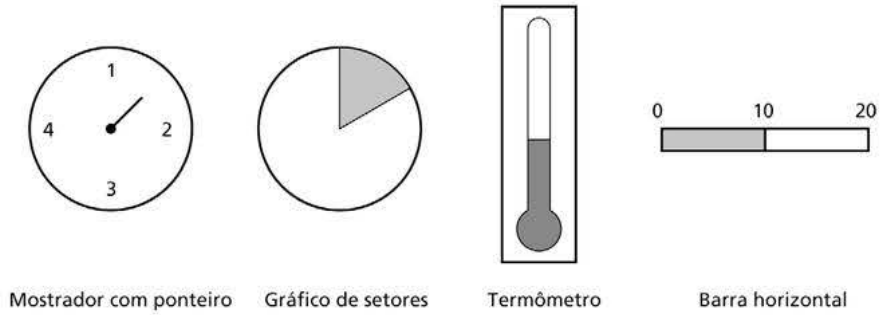
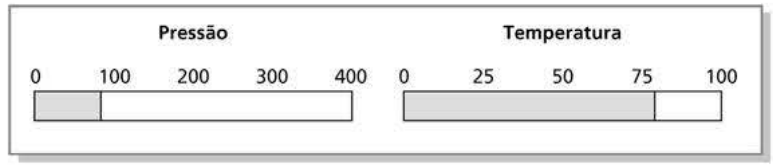


Figura 16.7

Display de informações gráficas que mostram valores relativos.



seu valor máximo, ao passo que a pressão não atingiu 25 por cento de seu valor máximo. Com um valor digital somente, o observador deve conhecer os valores máximos e calcular mentalmente o estado relativo da leitura. O tempo extra de raciocínio necessário pode levar a erros humanos em situações estressantes, quando os problemas ocorrem e os displays do operador podem mostrar leituras anormais.

Quando grandes quantidades de informações devem ser apresentadas, podem ser usadas visualizações abstratas que ligam itens de dados relacionados. Isso pode expor relacionamentos que não são óbvios com base em dados brutos. Você deve estar consciente das possibilidades de visualização, especialmente quando a interface com o usuário do sistema deve representar entidades físicas. Exemplos de visualizações de dados são:

1. Informações meteorológicas, obtidas de uma série de fontes, são mostradas como um mapa meteorológico com linhas isobáricas, frentes climáticas etc.
2. O estado de uma rede telefônica é exibido graficamente como um conjunto de nós ligados em um centro de gerenciamento de rede.
3. O estado de uma indústria química é visualizado ao mostrar as pressões e as temperaturas em um conjunto de tanques e tubulações ligadas.
4. O modelo de uma molécula é exibido e manipulado em três dimensões usando um sistema de realidade virtual.
5. Um conjunto de páginas Web é exibido como uma árvore hiperbólica (Lamping, et al., 1995).

Shneiderman (Shneiderman, 1998) oferece uma boa visão geral das abordagens para visualização, bem como identifica as classes de visualização que podem ser usadas. Essas classes incluem dados de visualização que usam apresentações de

duas e três dimensões, e como árvores e redes. A maioria dessas classes está relacionada à exibição de grandes quantidades de informações gerenciadas em um computador. Contudo, o uso mais comum de visualização nas interfaces com o usuário é representar algum tipo de estrutura física, como a estrutura molecular de um novo medicamento, os links em uma rede de telecomunicações etc. As apresentações em três dimensões, que podem usar equipamento especial de realidade virtual, são particularmente eficazes nas visualizações de produtos. A manipulação direta dessas visualizações é uma maneira eficaz de interagir com os dados.

Além do estilo de apresentação de informações, você deve pensar cuidadosamente sobre como as cores são usadas na interface. As cores podem aprimorar as interfaces com o usuário auxiliando-o a compreender e gerenciar a complexidade. Contudo, é fácil usar inadequadamente as cores e criar interfaces com o usuário que não sejam visualmente atraentes e sejam propensas a erros. Shneiderman apresenta 14 diretrizes principais para o uso eficiente das cores em interfaces com o usuário. As mais importantes dessas diretrizes são:

1. *Limitar o número de cores usadas e ser conservador quanto ao modo de serem utilizadas.* Você não deve usar mais do que quatro ou cinco cores separadas em uma janela e não mais do que sete em uma interface de sistema. Se você usar muitas cores ou se estas forem muito brilhantes, o display pode ficar confuso. Alguns usuários podem considerar que grandes quantidades de cores incomodam e são visualmente cansativas. A confusão do usuário também é possível se as cores forem usadas inconsistentemente.
2. *Usar a mudança de cores para mostrar uma mudança no status do sistema.* Se um display mudar de cor, isso deve significar que um evento significativo ocorreu. Portanto, em um mostrador de combustível, você pode usar uma mudança de cor para indicar que o tanque de combustível está ficando vazio. O destaque de cores é particularmente importante em displays complexos, nos quais centenas de entidades distintas podem ser exibidas.
3. *Usar o código de cores para apoiar a tarefa que os usuários estão tentando executar.* Se eles devem identificar instâncias anômalas, destaque essas instâncias; se as similaridades também devem ser descobertas, destaque essas similaridades usando uma cor diferente.
4. *Usar códigos de cores de maneira cuidadosa e consistente.* Se uma parte de um sistema exibe mensagens de erro em vermelho (por exemplo), todas as outras partes devem fazer o mesmo. A cor vermelha não deve ser usada para outra finalidade. Se for, o usuário pode interpretar o display vermelho como uma mensagem de erro.
5. *Ser cuidadoso quanto aos pares de cores.* Devido à fisiologia do olho humano, as pessoas não podem enfocar o vermelho e o azul simultaneamente. O estresse do olho é a provável conseqüência de um display vermelho sobre o azul. Outras combinações de cores podem também ser visualmente incômodas ou difíceis de ler.

Em geral, você deve usar cores para destacar, mas não deve associar significados a determinadas cores. Aproximadamente dez por cento dos homens são daltônicos e podem interpretar erroneamente o significado. As percepções humanas quanto às cores são diferentes, e existem convenções diferentes em profissões diferentes sobre o significado de determinadas cores. Os usuários com experiências diferentes podem, inconscientemente, interpretar a mesma cor de modo diverso. Por exemplo, para um motorista, vermelho significa geralmente *perigo*. Contudo, para um químico, vermelho significa *quente*.

Assim como a apresentação de informações da aplicação, os sistemas também se comunicam com os usuários por meio de mensagens que apresentam informações sobre os erros e o estado do sistema. Uma primeira experiência do usuário com um sistema de software pode ser quando o sistema apresenta uma mensagem de erro. Os usuários inexperientes podem começar a trabalhar, cometer um erro inicial e devem imediatamente compreender a mensagem de erro resultante. Isso pode ser difícil mesmo para engenheiros de software experientes. Geralmente é impossível para os usuários de sistema inexperientes ou casuais. Os fatores a serem levados em consideração ao projetar mensagens de sistema são mostrados na Tabela 16.3.

Você deve prever o conhecimento e a experiência dos usuários ao projetar mensagens de erro. Por exemplo, digamos que um usuário do sistema seja uma enfermeira em uma unidade de tratamento intensivo de um hospital. A monitoração do paciente é realizada por um sistema de computador. Para visualizar o estado atual do paciente (freqüência cardíaca, temperatura etc.), a enfermeira seleciona 'display' em um menu e insere o nome do paciente, conforme mostrado na Figura 16.8.

Nesse caso, vamos supor que a enfermeira errou o nome do paciente e digitou 'MacDonald' em vez de 'McDonald'. O sistema gerará uma mensagem de erro. As mensagens de erro devem ser sempre gentis, concisas, consistentes e construtivas. Elas não devem ser abusivas e não devem ter sinais de 'beep' associados ou outros ruídos que possam atrapalhar o usuário. Sempre que possível, a mensagem deve sugerir como o erro pode ser corrigido. A mensagem de erro deve estar ligada a um sistema de ajuda on-line sensível ao contexto.

A Figura 16.9 mostra exemplos de mensagens de erro boas e ruins. A mensagem do lado esquerdo foi projetada inadequadamente. Ela é negativa (acusa o usuário de cometer um erro), não está ajustada ao nível de habilidade e experiência do usuário e não leva em consideração as informações de contexto. Não sugere como a situação pode ser corrigida. Usa termos específicos do sistema (*id* de paciente) em vez de uma linguagem orientada a usuários. A mensagem do lado direito

Tabela 16.3 Fatores de projeto na redação de mensagens

Fator	Descrição
Contexto	Sempre que possível, as mensagens geradas pelo sistema devem refletir o contexto atual do usuário. Tanto quanto possível, o sistema deve estar consciente do que o usuário está fazendo e deve gerar mensagens relevantes a sua atividade atual.
Experiência	À medida que os usuários se tornam familiarizados com o sistema, eles ficam irritados com mensagens longas e explicativas. Entretanto, os iniciantes encontram dificuldades em compreender instruções curtas e sucintas de um problema. Você deve fornecer ambos os tipos de mensagens para permitir que o usuário controle a concisão delas.
Nível de habilidade	As mensagens devem ser ajustadas às habilidades do usuário, bem como a sua experiência. As mensagens para diferentes categorias de usuários podem ser expressas de maneiras diferentes, dependendo da terminologia familiar ao leitor.
Estilo	As mensagens devem ser positivas, e não negativas. Elas devem usar o modo ativo, e não passivo, de expressão. Nunca devem ser ofensivas ou engraçadas.
Cultura	Sempre que possível, o projetista de mensagens deve estar familiarizado com a cultura do país onde o sistema é vendido. Existem diferenças culturais distintas entre Europa, Ásia e América. Uma mensagem adequada para uma cultura pode ser inaceitável em outra.

Figura 16.8

Caixa de entrada de texto usada por uma enfermeira.

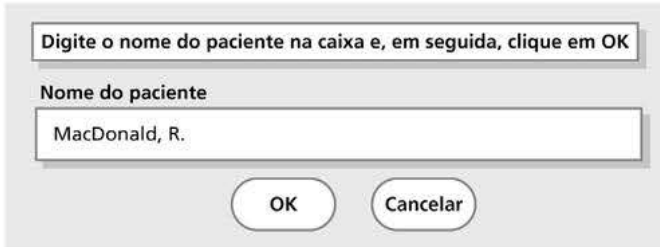
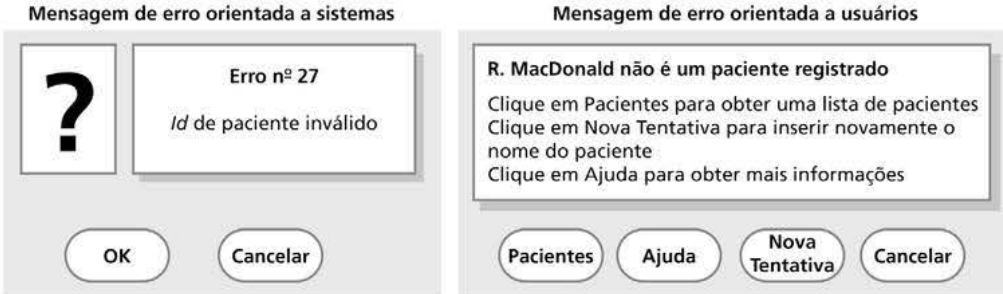


Figura 16.9

Mensagens de erro orientada a sistemas e orientada a usuários.



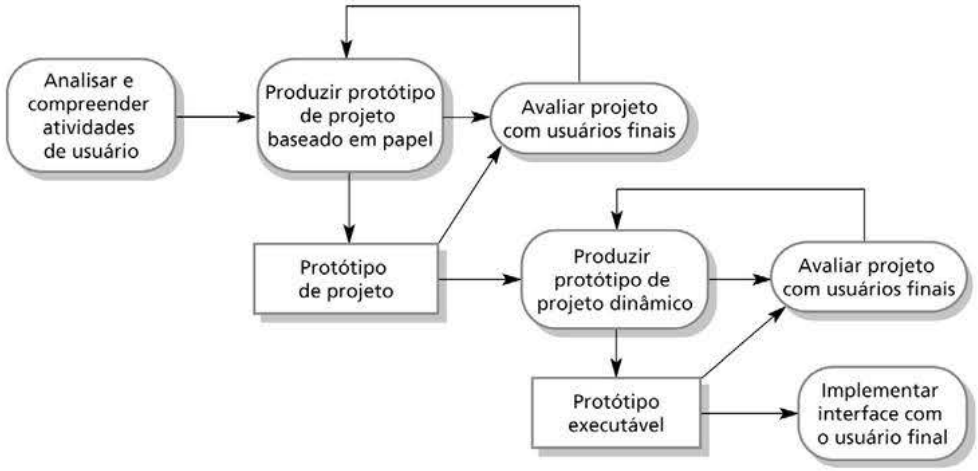
é melhor. Ela é positiva; indica que o problema é do sistema e não do usuário. Identifica o problema usando termos de enfermagem e oferece uma maneira fácil para corrigir o erro pressionando um único botão. O sistema de ajuda está disponível, caso seja necessário.

16.2 Processo de projeto de UI

A interface com o usuário (UI — *User Interface*) é um processo iterativo no qual os usuários interagem com os projetistas e os protótipos de interface para decidir sobre as características, a organização, o *look and feel* da interface com o usuário do sistema. Algumas vezes, o protótipo da interface é desenvolvido separadamente, em paralelo, com outras atividades de engenharia de software. Mais comumente, em especial quando o desenvolvimento iterativo é usado, o projeto de interface com o usuário prossegue incrementando-se, à medida que o software é desenvolvido. Em ambos os casos, contudo, antes de iniciar a programação, você deve ter desenvolvido e, de preferência, testado alguns projetos baseados em papel.

O processo geral de UI é ilustrado na Figura 16.10. Existem três atividades principais nesse processo:

Figura 16.10
Processo de projeto de UI.



1. *Análise de usuário.* No processo de análise de usuário, você desenvolve uma compreensão das tarefas que os usuários realizam, seu ambiente de trabalho, os outros sistemas que eles usam, como eles interagem com outras pessoas em seu trabalho etc. Para produtos com vários usuários, você deve tentar desenvolver essa compreensão por meio do foco em grupos, ensaios com usuários potenciais e exercícios similares.
2. *Prototipação de sistema.* O projeto e desenvolvimento da interface com o usuário é um processo iterativo. Embora os usuários possam informar sobre os recursos de interface de que necessitam, é muito difícil para eles serem específicos até que vejam algo tangível. Portanto, você deve desenvolver sistemas de protótipo e expô-los aos usuários, que poderão, então, guiar a evolução da interface.
3. *Avaliação de interface.* Embora haja, obviamente, discussões com os usuários durante o processo de prototipação, você deve também ter uma atividade de avaliação mais formalizada, na qual sejam coletadas informações sobre a experiência do usuário com a interface.

Enfoco a análise de usuário e avaliação de interface nesta seção com apenas uma breve explicação das técnicas específicas de prototipação de interface com o usuário. Abordo questões mais gerais de prototipação e suas técnicas no Capítulo 17.

O cronograma de projeto de UI dentro do processo de software depende, em alguma extensão, de outras atividades. Conforme explicado no Capítulo 7, a prototipação pode ser usada como parte do processo de engenharia de requisitos e, nesse caso, faz sentido iniciar o processo de projeto de UI por esse estágio. Nos processos iterativos, explicados no Capítulo 17, o projeto de UI é integrado ao desenvolvimento de software. Como o software propriamente dito, a UI deve passar por refactoring e reprojeto durante o desenvolvimento.

16.3 Análise de usuário

Uma atividade crítica de projeto de UI é a análise das atividades de usuário apoiadas pelo sistema de computador. Se você não compreender o que os usuários querem fazer com o sistema, você não terá uma perspectiva realista do projeto de uma interface com o usuário eficiente. Para desenvolver essa compreensão, você pode usar técnicas, como a análise de tarefas, estudos etnográficos, interfaces com o usuário e observações ou, mais comumente, uma combinação de todas essas técnicas.

Um desafio para os engenheiros envolvidos na análise de usuário é encontrar uma maneira de descrever as análises, de modo que eles comuniquem a essência das tarefas a outros projetistas e aos usuários em si. As notações, como os diagramas de seqüência da UML, podem ser úteis para descrever as interações do usuário e são ideais para a comunicação com os engenheiros de software. Contudo, outros usuários podem considerar esses diagramas muito técnicos e desistirão de compreendê-los. Devido à importância do envolvimento dos usuários no processo de projeto, geralmente é necessário desenvolver cenários em linguagem natural para descrever as atividades do usuário.

O Quadro 16.1 é um exemplo de cenário em linguagem natural que poderia ter sido desenvolvido durante o processo de especificação e projeto para o sistema LIBSYS. Ele descreve uma situação em que o LIBSYS não existe e uma estudante necessita recuperar informações de outra biblioteca. A partir desse cenário, o projetista pode verificar uma série de requisitos:

Quadro 16.1

Cenário de interação de biblioteca.

Jane é uma estudante de assuntos religiosos que está escrevendo um ensaio sobre a arquitetura da Índia e como esta foi influenciada pelas práticas religiosas. Para ajudar em sua compreensão, ela gostaria de acessar figuras detalhadas sobre edifícios monumentais, mas não consegue encontrar algo na sua biblioteca local. Ela se dirige ao bibliotecário da área para explicar suas necessidades e ele sugere termos de busca que ela possa usar. Ele sugere também bibliotecas em Nova Délhi e Londres que possam ter este material e, assim, ele e Jane se conectam aos catálogos da biblioteca e realizam uma busca usando esses termos. Encontram material básico e enviam uma solicitação de fotocópias das figuras com detalhes de arquitetura a serem enviadas, via correio, diretamente para Jane.

1. Os usuários podem não estar conscientes dos termos de busca apropriados. Pode ser necessário acessar ajuda quanto à escolha de termos de busca.
2. Os usuários precisam ser capazes de selecionar conjuntos para busca.
3. Os usuários precisam ser capazes de realizar buscas e solicitar cópias do material relevante.

Você não deve esperar pela análise de usuário para gerar requisitos muito específicos de interface com o usuário. Normalmente, a análise ajuda a compreender as necessidades e assuntos dos usuários do sistema. Ao tornar-se mais consciente de como eles trabalham, de seus assuntos e suas restrições, o projeto pode passar a considerar essas questões. Isso significa que seus projetos iniciais (a serem refinados por meio de alguma prototipação) serão, provavelmente, mais aceitos pelos usuários, sendo mais fácil convencê-los a se envolverem no processo de refinamento do projeto.

16.3.1 Técnicas de análise

Conforme expliquei na seção anterior, existem três técnicas básicas de análise de usuário: análise de tarefas, entrevistas e questionários e etnografia. A análise de tarefa e a entrevista enfocam o indivíduo e o trabalho individual, sendo que a etnografia abrange uma perspectiva mais ampla e estuda como as pessoas interagem umas com as outras, como elas organizam seu ambiente de trabalho e cooperam entre si para resolver os problemas.

Existem várias formas de análise de tarefa (Diaper, 1989), mas a mais comumente usada é a Análise de Tarefas Hierárquicas (HTA — *Hierarchical Task Analysis*). A HTA foi originalmente desenvolvida para auxiliar a escrita de manuais de usuário, mas pode também ser usada para identificar o que os usuários fazem para atingir algum objetivo. Em HTA, uma tarefa de nível alto é dividida em subtarefas, e os planos, que especificam o que poderia ocorrer em uma situação específica, são identificados. Começando com um objetivo de usuário, você deve desenhar uma hierarquia que mostra o que deve ser feito para atingir esse objetivo. A Figura 16.11 ilustra essa abordagem usando o cenário de biblioteca apresentado no Quadro 16.1. Na notação em HTA, a linha sob a caixa de tarefa indica, normalmente, que a tarefa será decomposta em subtarefas mais detalhadas.

A vantagem da HTA em relação aos cenários em linguagem natural é que ela o força a considerar cada uma das tarefas e a decidir se elas devem ser decompostas. Com cenários em linguagem natural, é fácil ignorar tarefas importantes. Os cenários tornam-se também longos e maçantes de serem lidos, caso você deseje lhes adicionar grande quantidade de detalhes.

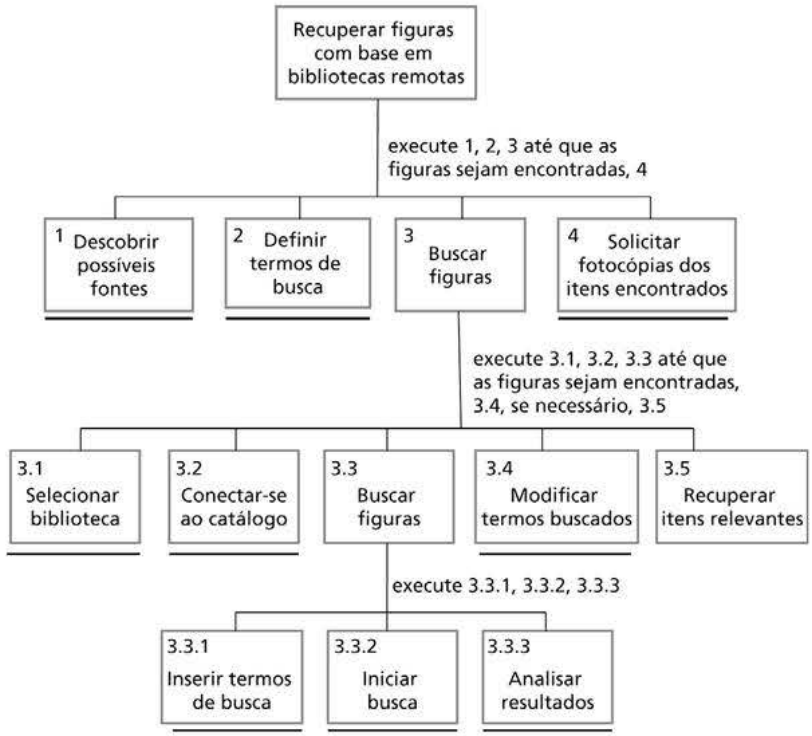
O problema com essa abordagem para descrever tarefas de usuário é que ela é mais adequada se as tarefas forem seqüenciais. A notação torna-se incômoda quando você tentar modelar tarefas que envolvem atividades intercaladas ou concorrentes ou que envolvem um número muito grande de subtarefas. Por isso, a HTA não registra por que as tarefas são feitas de determinada maneira ou as restrições sobre os processos de usuário. Você pode obter uma visão parcial das atividades de usuário com base na HTA, mas são necessárias informações adicionais para desenvolver uma compreensão mais completa dos requisitos de projeto de UI.

Normalmente, você coleta informações para a HTA por meio de observações e entrevistas com usuários. Nesse processo de entrevista, você pode coletar algumas dessas informações adicionais e registrá-las junto com a análise da tarefa. Para descobrir o que os usuários realmente fazem, você deve projetar as entrevistas de modo que os usuários possam fornecer alguma informação que eles (e não você) julgam ser relevante. Isso significa que você não deve se manter rigidamente na lista de questões preparadas. Em vez disso, suas questões devem ser abertas e encorajar os usuários a contar como realizam as tarefas bem como o que realmente fazem.

A entrevista, naturalmente, não é apenas uma maneira de obter informações para a análise de tarefa — é uma técnica geral de coleta de informações. Você pode decidir complementar as entrevistas individuais com entrevistas em grupo ou grupos específicos. A vantagem do uso de grupos específicos é que os usuários estimulam uns aos outros a fornecer informações e podem ser discutidas maneiras diferentes de usar o sistema.

Figura 16.11

Análise de tarefas hierárquicas.



A análise de tarefa enfoca como as pessoas trabalham, mas, naturalmente, a maior parte do trabalho é cooperativo. As pessoas trabalham juntas para atingir um objetivo e os usuários consideram difícil discutir como essa cooperação realmente acontece. Portanto, a observação direta de como os usuários trabalham e usam os sistemas baseados em computador é uma técnica adicional importante para a análise de usuário.

Uma abordagem para observação direta usada em uma ampla variedade de locais é a etnografia (Suchman, 1983; Hughes, et al., 1997; Crabtree, 2003). Expliquei a etnografia no Capítulo 7 como uma técnica que apóia a engenharia de requisitos. Os etnógrafos observam atentamente como as pessoas trabalham, como interagem com os outros e como os recursos no espaço de trabalho são usados para dar suporte ao trabalho. A vantagem da etnografia é que o etnógrafo pode observar ações intuitivas e colaborações informais que podem, por isso, iniciar discussões posteriores sobre o trabalho.

Como exemplo da maneira que a etnografia pode influenciar o projeto de interface com o usuário, o Quadro 16.2 é um fragmento de um relatório de estudo etnográfico sobre controladores de tráfego aéreo no qual estive envolvido (Bentley, et al., 1992). Estávamos interessados no projeto de interface para um sistema de ATC mais automatizado e aprendemos duas coisas importantes tendo como base estas observações:

1. Os controladores tinham de ser capazes de ver todos os vôos em um setor (isso porque eles espalhavam as fitas sobre a mesa). Portanto, devíamos evitar o uso de displays com scrolling nos quais os vôos desapareceriam da parte superior e inferior do display.
2. A interface devia ter alguma maneira de informar aos controladores quantos vôos estavam em setores adjacentes, de modo que eles pudessem planejar sua carga de trabalho.

Quadro 16.2

Relatório de observações de controle de tráfego aéreo.

O controle de tráfego aéreo envolve uma série de equipamentos de controle, sendo que os equipamentos que controlam setores adjacentes do espaço aéreo estão fisicamente localizados próximos uns dos outros. Os vôos em um setor são representados pelas fitas de papel colocadas em estruturas de madeira, a fim de refletirem sua posição no setor. Se não houver espaços suficientes na estrutura (por exemplo, quando o espaço aéreo estiver muito ocupado), os controladores espalharão as fitas sobre a mesa em frente das estruturas de madeira. Enquanto observávamos os controladores, notamos que eles regularmente olhavam de relance para as estruturas de fitas no setor adjacente. Comentamos isso com eles e perguntamos por que faziam isso. Eles responderam que, quando o controlador adjacente tem fitas em sua mesa, isso significa que grande quantidade de vôos vai entrar em seu setor. Eles, portanto, tentavam aumentar a velocidade da aeronave no setor para 'limpar o espaço' para a aeronave que estava chegando.

A verificação de setores adjacentes era uma ação automática do controlador, e é muito provável que não tivessem mencionado isso nas discussões de processo de ATC. Foi somente por meio da observação direta que descobrimos esses requisitos importantes.

Nenhuma das técnicas de análise de usuário, por si mesmas, apresentam um quadro completo do que os usuários realmente fazem. Elas são abordagens complementares que você deve usar em conjunto para ajudar a compreender o que os usuários fazem e obter visões do que pode ser um projeto adequado de interface com o usuário.

16.4 Prototipação de interface com o usuário

Devido à natureza dinâmica das interfaces com o usuário, as descrições textuais e diagramas não são boas o suficiente para expressar requisitos de interface com o usuário. A prototipação evolucionária ou exploratória, com o envolvimento dos usuários finais, é a única maneira prática de projetar e desenvolver interfaces gráficas com o usuário para sistemas de software. O envolvimento do usuário no processo de projeto e desenvolvimento é um aspecto essencial do *projeto centrado em usuários* (Norman e Draper, 1986), uma filosofia de projeto para sistemas interativos.

O objetivo da prototipação é permitir que os usuários ganhem experiência direta com a interface. A maioria de nós considera difícil pensar de maneira abstrata sobre uma interface com o usuário para explicar exatamente o que queremos. Entretanto, quando são apresentados exemplos, é fácil identificar as características de que gostamos ou não.

De preferência, quando estiver desenvolvendo um protótipo de uma interface com o usuário, você deve adotar um processo de prototipação de dois estágios:

1. Bem no início do processo, você deve desenvolver protótipos de papel — maquetes de projetos de tela — e repassar o processo com os usuários finais tendo como base esses protótipos.
2. Depois você deve refinar seu projeto e desenvolver protótipos sofisticados automatizados gradativamente para, então, torná-los disponíveis aos usuários para testes e simulação de atividades.

A prototipação de papel é uma abordagem barata e surpreendemente eficaz no desenvolvimento de protótipos (Snyder, 2003). Você não necessita desenvolver qualquer software executável e os projetos não devem ser elaborados com padrões profissionais. Você pode desenhar versões em papel das telas do sistema com as quais os usuários interagem e projetar um conjunto de cenários que descrevam como o sistema pode ser usado. À medida que um cenário progride, você deve esboçar as informações exibidas e as opções disponíveis aos usuários.

Portanto, trabalhe com esses cenários com os usuários para simular como o sistema pode ser usado. Essa é uma maneira eficaz de observar as reações iniciais do usuário a um projeto de interface, as informações de que ele necessita do sistema e como ele interagiria normalmente com o sistema.

Como alternativa, você pode usar uma técnica de storyboarding para apresentar o projeto de interface. Um storyboard é uma série de esboços que ilustram uma seqüência de interações. Isso é menos prático, mas pode ser mais conveniente ao apresentar propostas de interfaces para grupos em vez de para indivíduos.

Após experimentos iniciais com um protótipo de papel, você deve implementar um protótipo de software do projeto de interface. O problema, naturalmente, é que você precisa ter alguma funcionalidade de sistema com a qual os usuários possam interagir. Se estiver desenvolvendo um protótipo de UI logo no início do processo de desenvolvimento do sistema, essa funcionalidade pode não estar disponível. Para contornar esse problema, você pode usar a prototipação 'Mágico de Oz'. Nessa abordagem, os usuários interagem com o que parece ser um sistema de computador, mas suas entradas são na verdade direcionadas a uma pessoa oculta que simula as respostas do sistema. Eles podem fazer isso diretamente ou por meio de algum outro sistema para calcular as respostas necessárias.

Os experimentos posteriores de protótipo podem, então, ser realizados com o uso de uma abordagem evolucionária ou throw-away. Explico essas abordagens de prototipação no Capítulo 17, no qual descrevo também uma variedade de técnicas que podem ser usadas para prototipação e desenvolvimento rápido de aplicações. Existem três abordagens que podem ser usadas para a prototipação de interface com o usuário:

1. *Abordagem dirigida a scripts.* Se você precisa simplesmente explorar idéias com os usuários, você pode usar uma abordagem dirigida a scripts, como você encontraria no Macromedia Director. Nessa abordagem, você cria telas com elementos visuais, como botões e menus, e associa um script a esses elementos. Quando o usuário interage com essas telas, o script é executado e a próxima tela é apresentada, mostrando nas telas os resultados de suas ações. Não existe lógica de aplicação envolvida.
2. *Linguagens de programação visual.* As linguagens de programação visual, como Visual Basic, incorporam um ambiente de desenvolvimento poderoso, acessam uma variedade de objetos reusáveis e um sistema de desenvolvimento de

interface com o usuário que permite às interfaces serem criadas rapidamente, com componentes e scripts associados aos objetos de interface. Descrevo sistemas de desenvolvimento visual no Capítulo 17.

3. *Prototipação baseada na Internet*. Essas soluções, baseadas em navegadores Web e linguagens como Java, oferecem uma interface pronta com o usuário. Você pode adicionar funcionalidades por associação de segmentos de programas em Java com as informações a serem exibidas. Esses segmentos (chamados applets) são executados automaticamente quando a página é carregada em um navegador. Essa abordagem é uma maneira rápida de desenvolver protótipos de interfaces com o usuário, mas existem restrições inerentes impostas pelo navegador e pelo modelo de proteção Java.

A prototipação está obviamente bastante associada à avaliação de interfaces. A avaliação formal não é provavelmente adequada em custo aos protótipos iniciais e, assim, o objetivo desse estágio é uma 'avaliação formativa' na qual você procura maneiras de aprimorar as interfaces. À medida que o protótipo torna-se mais completo, podem-se usar técnicas de avaliação sistemática, conforme será explicado na próxima seção.

16.5 Avaliação de interface

A avaliação de interface é o processo de avaliação de usabilidade de uma interface e de verificação de se a interface atende aos requisitos de usuário. Portanto, deve ser parte do processo normal de verificação e validação de sistemas de software. Neilsen (Neilsen, 1993) inclui um bom capítulo sobre esse assunto em seu livro sobre engenharia de usabilidade.

Preferencialmente, uma avaliação deve ser conduzida em relação a uma especificação baseada nos atributos de usabilidade, conforme mostrado na Tabela 16.4. As métricas desses atributos de usabilidade podem ser criadas. Por exemplo, em uma especificação de facilidade de aprendizado, você pode estabelecer que um operador, familiarizado com o trabalho apoiado, deve ser capaz de usar 80 por cento da funcionalidade de sistema, depois de uma sessão de treinamento de três horas. Contudo, é mais comum especificar a usabilidade (caso seja especificada) qualitativamente em vez de usar métricas. Você, portanto, deve usar em geral seu julgamento e experiência na avaliação de interfaces.

A avaliação sistemática de um projeto de interface com o usuário pode ser um processo dispendioso que envolve cientistas de cognição e projetistas gráficos. Você pode precisar projetar e realizar uma série de experimentos significativos com usuários típicos. Pode ser necessário usar laboratórios especialmente construídos com equipamento de monitoração. Uma avaliação de interface com o usuário desse tipo é economicamente irreal para sistemas desenvolvidos por pequenas organizações com recursos limitados.

Existe uma série de técnicas mais simples e menos dispendiosa de avaliação de interface com o usuário que podem identificar deficiências específicas de projeto de interface com o usuário:

1. Questionários que coletem informações sobre o que os usuários pensam das interfaces.
2. Observar usuários no trabalho com o sistema e imaginar como eles estão tentando usar o sistema para realizar alguma tarefa.
3. Fotos e vídeo do uso típico do sistema.
4. A inclusão de código que coleta informações sobre os recursos mais usados e os erros mais comuns no software.

Pesquisar usuários por meio de questionário é uma maneira relativamente barata para avaliar a interface. As questões devem ser precisas, e não genéricas. Não use questões como 'Comente a usabilidade da interface', pois as respostas, provavelmente, variarão tanto que você não constatará nenhuma tendência comum. Antes disso, questões específicas, como 'Avalie a facilidade de compreensão de mensagens de erro usando uma escala de 1 a 5. O valor 1 significa muito claro e 5

Tabela 16.4 Atributos de usabilidade

Atributo	Descrição
Facilidade de aprendizado	Quanto tempo leva para que um novo usuário torne-se produtivo com o sistema?
Velocidade de operação	Qual a adequação da resposta do sistema com a prática de trabalho do usuário?
Robustez	Quão tolerante é o sistema em relação aos erros do usuário?
Facilidade de recuperação	Quão bom é o sistema quanto à recuperação de erros do usuário?
Facilidade de adaptação	Quão fielmente o sistema está ligado a um único modelo de trabalho?

significa incompreensível', são melhores. Eles são mais fáceis de responder e a probabilidade de que forneçam informações úteis para melhorar o software é maior.

Você deve solicitar aos usuários que avaliem suas próprias experiências e conhecimentos ao preencherem o questionário. Isso permite ao projetista descobrir se os usuários com determinado conhecimento têm problemas com a interface. Os questionários podem ser usados antes mesmo que qualquer sistema executável esteja disponível, caso uma maquete de papel da interface seja construída e avaliada.

A avaliação baseada em observação envolve simplesmente assistir aos usuários enquanto eles utilizam o sistema, observando os recursos usados, os erros cometidos etc. Isso pode ser complementado por sessões de 'pense alto' nas quais os usuários falam sobre o que estão tentando conseguir, como eles compreendem o sistema e como estão tentando usar o sistema para realizar seus objetivos.

Um equipamento de vídeo de custo relativamente baixo possibilita registrar sessões de usuário para análise posterior. A análise completa de vídeo é dispendiosa e requer um conjunto de avaliação especialmente equipado com várias câmeras focalizadas no usuário e na tela. Entretanto, o registro de vídeo de operações selecionadas do usuário pode ser útil na detecção de problemas. Outros métodos de avaliação devem ser usados para descobrir quais operações causam dificuldades ao usuário.

A análise de registros permite ao projetista descobrir se a interface requer muitos movimentos manuais (um problema com alguns sistemas é que os usuários devem mover regularmente sua mão do teclado para o mouse) e verificar se são necessários movimentos não naturais dos olhos. Uma interface que exija muitos movimentos dos olhos pode levar o usuário a cometer mais erros ou ignorar partes do display.

A instrumentação de código para coletar estatísticas de uso permite que as interfaces sejam aprimoradas de várias maneiras. As operações mais comuns podem ser detectadas. A interface pode ser reorganizada de modo que essas operações sejam mais rápidas de selecionar. Por exemplo, se menus do tipo pop-up e pull-down forem usados, as operações mais freqüentes devem estar na parte superior do menu, e as operações destrutivas, mais abaixo. A instrumentação de código também permite que comandos propensos a erros sejam detectados e modificados.

Finalmente, é fácil apresentar aos usuários um comando 'gripe' que eles podem usar para passar mensagens ao projetista da ferramenta. Isso faz com que os usuários sintam que seu ponto de vista está sendo considerado. O projetista da interface e outros engenheiros podem obter feedback rápido sobre problemas específicos.

Nenhuma dessas abordagens relativamente simples para avaliação de interface com o usuário é à prova de falhas, e elas, provavelmente, não detectam todos os problemas. Contudo, as técnicas podem ser usadas com um grupo de voluntários antes que um sistema seja liberado sem grande dispêndio de recursos. Muitos dos piores problemas de projeto de interface com o usuário podem, assim, ser descobertos e corrigidos.

PONTOS-CHAVE

- Os princípios de interface com o usuário que abordam familiaridade, consistência, fator surpresa mínimo, facilidade de recuperação, guia de usuário e diversidade de usuário ajudam a guiar o projeto de interfaces com o usuário.
- Estilos de interação com um sistema de software incluem manipulação direta, sistemas de menus, preenchimento de formulários, linguagens de comando e linguagem natural.
- O display de informações gráficas deve ser usado quando for dedicado a apresentar as tendências e valores aproximados. O display digital deve ser somente usado quando for necessária uma precisão maior.
- As cores devem ser usadas cuidadosa e consistentemente em interfaces com o usuário. Os projetistas devem levar em consideração o fato de que um número significativo de pessoas são daltônicas.
- O projeto de interface com o usuário inclui subprocessos relacionados à análise de usuário, prototipação de interfaces e avaliação de interfaces.
- O objetivo da análise de usuário é sensibilizar os projetistas quanto aos meios pelos quais os usuários realmente trabalham. Você deve usar técnicas diferentes — análise de tarefas, entrevistas e observações — durante a análise de usuário.
- O desenvolvimento de protótipos de interface com o usuário deve ser um processo por estágios com os protótipos iniciais baseados em versões em papel que, depois da avaliação inicial e feedback, são usadas como base para protótipos automatizados.
- Os objetivos da avaliação de interface com o usuário são obter feedback de como um projeto de UI pode ser aprimorado e avaliar se uma interface atende aos requisitos de usabilidade.

LEITURAS SUGERIDAS

Human-computer interaction, 3rd ed. Um bom texto cujos pontos fortes são o enfoque nas questões de projeto e o trabalho cooperativo. (A. Dix, et al., 2004, Prentice Hall.)

Interaction design. O foco deste livro recai sobre o projeto de interação com sistemas baseados em computador. Ele aborda o mesmo assunto de *Human-computer interation*, mas de maneira bem diferente. Ambos os livros são bem escritos e vale a pena sua leitura. (J. Preece, et al., 2002, John Wiley & Sons.)

"Usability engineering". Esta edição especial do *IEEE software* inclui uma série de artigos sobre usabilidade escritos especialmente para os leitores com algum conhecimento sobre engenharia de software. (*IEEE software*, 18(1), janeiro de 2001.)

EXERCÍCIOS

16.1 Sugeri na Seção 16.1 que os objetos manipulados pelos usuários devam ser retirados de seu domínio em vez do domínio do computador. Sugira objetos apropriados para os seguintes usuários e sistemas.

- Um ajudante de almoxarifado que usa um catálogo automatizado de peças
- Um piloto de uma linha aérea que usa um sistema de monitoração de segurança de aeronaves
- Um gerente que manipula um banco de dados financeiros
- Um policial que usa um sistema de controle de viatura

16.2 Sugira situações nas quais é muito complicado ou impossível fornecer uma interface consistente com o usuário.

16.3 Quais fatores precisam ser levados em consideração no projeto de uma interface baseada em menus para sistemas 'walk-up', como os caixas eletrônicos de bancos? Escreva um comentário crítico sobre a interface de um caixa eletrônico que você usa.

16.4 Sugira maneiras pelas quais a interface com o usuário de um sistema de e-commerce, como uma livraria ou um site de música on-line, poderia ser adaptada aos usuários com dificuldades visuais ou problemas neurológicos.

16.5 Explique as vantagens do display de informações gráficas e sugira quatro aplicações em que seria mais apropriado usar displays gráficos, e não digitais, de informações numéricas.

16.6 Quais são as diretrizes a serem seguidas quando se usam cores em uma interface com o usuário? Sugira como a cor poderia ser usada mais eficazmente na interface de um sistema de aplicação que você utiliza.

16.7 Considere as mensagens de erro produzidas pelo MS-Windows, Linux, Mac OS ou outro sistema operacional. Sugira como elas poderiam ser aprimoradas.

16.8 Escreva cenários possíveis de interação para os seguintes sistemas:

- O uso de um serviço de reservas de teatro baseado na Web para compra de bilhetes de teatro e pagamento com cartão de crédito
- A compra dos mesmos bilhetes com utilização de interface de telefone celular
- O uso de conjunto de ferramenta CASE para criar um modelo de objeto de um sistema de software (veja os capítulos 8 e 14) e a geração automática de código baseada nesse modelo.

16.9 Sob quais circunstâncias você pode usar a prototipação 'Mágico de Oz'? Para quais tipos de sistemas essa abordagem é inadequada?

16.10 Crie um questionário para obter informações sobre a interface com o usuário de alguma ferramenta (como um processador de textos) com a qual você esteja familiarizado. Se possível, distribua esse questionário a uma série de usuários e tente avaliar os resultados. O que esses resultados informam sobre o projeto de interface com o usuário?

16.11 Explique se é ético instrumentar o software para monitorar seu uso sem informar aos usuários finais que seu trabalho está sendo monitorado.

16.12 Quais questões éticas os projetistas de interface com o usuário podem enfrentar ao tentar conciliar as necessidades de usuários finais de um sistema com as necessidades da organização que está pagando pelo desenvolvimento do sistema.

PARTE 4 DESENVOLVIMENTO

Quando a engenharia de software foi definida como uma disciplina, o processo de desenvolvimento da maioria dos sistemas era um processo de escrita de um programa baseado em uma especificação de projeto. Linguagens de programação imperativas como C, FORTRAN ou Ada foram usadas. Nos textos de engenharia de software, os capítulos sobre desenvolvimento de software, em sua maioria, enfocavam práticas de boa programação.

Atualmente há muitas maneiras diferentes de desenvolver software. Estas incluem programação básica em linguagens, como C++ ou Java, *scripting*, programação de banco de dados, geração de programas a partir de ferramentas CASE e engenharia de software baseada em reuso. Além disso, o fato de que não há uma distinção real entre desenvolvimento e manutenção está sendo finalmente reconhecido, e estamos começando a considerar o desenvolvimento como o primeiro estágio em um processo de evolução de programas. Para refletir esse desenvolvimento, incluí esta nova parte no livro, que enfoca as técnicas de desenvolvimento. Há cinco capítulos nesta parte:

1. O Capítulo 17 é um novo capítulo que descreve técnicas para o desenvolvimento rápido de software. No ambiente de negócios atual, as empresas necessitam que seus softwares sejam entregues rapidamente, de modo que elas possam responder aos novos desafios e oportunidades. Neste capítulo, explico métodos ágeis de desenvolvimento, com um enfoque especial sobre a *extreme programming*. Também descrevo ambientes para desenvolvimento rápido de aplicações e o uso adequado de prototipação de sistema.
2. O tópico dos capítulos 18 e 19 é a engenharia de software baseada em reuso. Nos últimos anos, o reuso de software tornou-se amplamente conhecido e o desenvolvimento baseado em reuso agora é uma abordagem importante para a engenharia de software. O Capítulo 18 apresenta uma visão geral do reuso e do desenvolvimento com reuso. O Capítulo 19 enfoca a engenharia de software baseada em componentes, incluindo a composição de componentes e o processo CBSE.
3. O Capítulo 20 continua a explicação sobre sistemas críticos que permeia o livro. Há uma gama de abordagens de desenvolvimento para atingir a confiança de sistemas, incluindo prevenção de defeitos e tolerância a defeitos, e explico como construções e técnicas de programação podem ser usadas para atingir essa confiança. Na parte final, retorno ao tópico de arquiteturas de software e descrevo abordagens de arquiteturas para tolerância a defeitos.
4. O Capítulo 21 é destinado à evolução de software. A mudança é inevitável para todos os sistemas de software e, em vez de considerar o processo de mudança como uma atividade separada, penso que faz sentido considerá-lo como uma continuação do desenvolvimento inicial do software. Nesse capítulo, explico a inevitabilidade da evolução, a manutenção de software, os processos de evolução e tomada de decisões em relação à evolução de sistemas legados.

17

Desenvolvimento rápido de software

Objetivos

O objetivo deste capítulo é descrever um número de abordagens para desenvolvimento de software que são dirigidas à entrega rápida de software. Depois de ler este capítulo, você:

- compreenderá como uma abordagem de software incremental e iterativa conduz a uma entrega mais rápida de software mais útil;
- conhecerá as diferenças entre métodos de desenvolvimento ágeis e métodos de desenvolvimento de software que contam com especificações e projetos documentados;
- aprenderá os princípios, práticas e algumas limitações da *extreme programming*;
- compreenderá como a prototipação pode ser usada para ajudar a resolver dúvidas a respeito de requisitos e de projeto quando uma abordagem de desenvolvimento baseada em especificações foi usada.

Conteúdo

17.1 Métodos ágeis
17.2 *Extreme programming*
17.3 Desenvolvimento rápido de aplicações
17.4 Prototipação de software

Os negócios atualmente operam em um ambiente global sujeito a rápidas mudanças. Eles têm de responder a novas oportunidades e mercados, mudanças de condições econômicas e ao surgimento de produtos e serviços concorrentes. O software é parte de quase todas as operações de negócio e, assim, é essencial que um novo software seja desenvolvido rapidamente para aproveitar as novas oportunidades e responder às pressões competitivas. Desenvolvimento e entrega rápidas são, portanto, muitas vezes o requisito mais crítico para sistemas de software. De fato, muitos negócios estão dispostos a equilibrar a qualidade e compromisso com requisitos do software com a entrega rápida.

Devido aos negócios estarem operando em um ambiente em ritmo de mudanças, muitas vezes é praticamente impossível derivar um conjunto completo de requisitos de software estáveis. Os requisitos que são propostos mudam inevitavelmente porque os clientes acham impossível prever

como um sistema afetará as práticas de trabalho, como interagirá com outros sistemas e quais operações de usuário devem ser automatizadas. Pode ser que somente após a entrega de um sistema e este ser experimentado pelos usuários é que os reais requisitos tornem-se claros.

Os processos de desenvolvimento de software baseados em especificações completas de requisitos, projeto, construção e teste de sistema não estão voltados para desenvolvimento rápido de software. Quando os requisitos mudam ou quando os problemas com requisitos são descobertos, o projeto de sistema ou a implementação precisa ser retrabalhada e retestada. Como conseqüência, um processo em cascata convencional ou baseado em especificações é geralmente prolongado e o software final é entregue para o cliente muito depois de ter sido originalmente especificado.

Em um ambiente de negócios de rápidas mudanças, isso pode causar problemas reais. No momento em que o software estiver disponível para uso, a razão original para sua aquisição pode ter mudado tão radicalmente que o software será inútil. Portanto, para sistemas de negócios específicos, processos de desenvolvimento voltados para o desenvolvimento e entrega rápidos são essenciais.

Processos de desenvolvimento rápido de software são projetados para criar software útil rapidamente. Geralmente, eles são processos iterativos nos quais a especificação, o projeto, o desenvolvimento e o teste são intercalados. O software não é desenvolvido e disponibilizado integralmente, mas em uma série de incrementos, e cada incremento inclui uma nova funcionalidade do sistema. Embora existam muitas abordagens para o desenvolvimento rápido de software, elas compartilham algumas características fundamentais:

1. Os processos de especificação, projeto e implementação são concorrentes. Não há especificação detalhada de sistema e a documentação de projeto é minimizada ou gerada automaticamente por um ambiente de programação usado para implementar o sistema. O documento de requisitos do usuário define somente as características mais importantes do sistema.
2. O sistema é desenvolvido em uma série de incrementos. Os usuários finais e outros stakeholders com o sistema participam da especificação e da avaliação de cada incremento. Eles podem propor alterações e novos requisitos que devem ser implementados em um incremento posterior do sistema.
3. As interfaces com o usuário do sistema são geralmente desenvolvidas usando-se um sistema de desenvolvimento interativo que permite que o projeto de interface seja criado rapidamente por desenho e inserção de ícones na interface. O sistema poderá, então, gerar uma interface baseada na Web para um navegador ou uma interface para uma plataforma específica, como o Microsoft Windows.

O desenvolvimento incremental, apresentado no Capítulo 4, envolve a produção e a entrega de software em incrementos, em vez de em um único pacote. Cada iteração do processo produz um novo incremento de software. As duas principais vantagens de adotar uma abordagem incremental para o desenvolvimento de software são:

1. *Entrega acelerada dos serviços ao cliente*. Os incrementos iniciais do sistema podem fornecer uma funcionalidade de alta prioridade, de forma que os clientes logo poderão obter valor do sistema durante seu desenvolvimento. Os clientes podem ver seus requisitos na prática e especificar mudanças para serem incorporadas nos releases posteriores do sistema.
2. *Engajamento do usuário com o sistema*. Os usuários do sistema precisam estar envolvidos no processo de desenvolvimento incremental porque eles devem dar feedback à equipe de desenvolvimento sobre os incrementos entregues. Esse envolvimento não significa somente que o sistema terá mais chances de atender aos requisitos, mas também que os usuários finais que estão comprometidos com o sistema querem vê-lo funcionando.

A Figura 17.1 ilustra um modelo de processo geral para desenvolvimento incremental. Note que os estágios iniciais desse processo enfocam o projeto da arquitetura. Se você não considerar a arquitetura no início do processo, a estrutura total do sistema será provavelmente instável e não comportará novos incrementos.

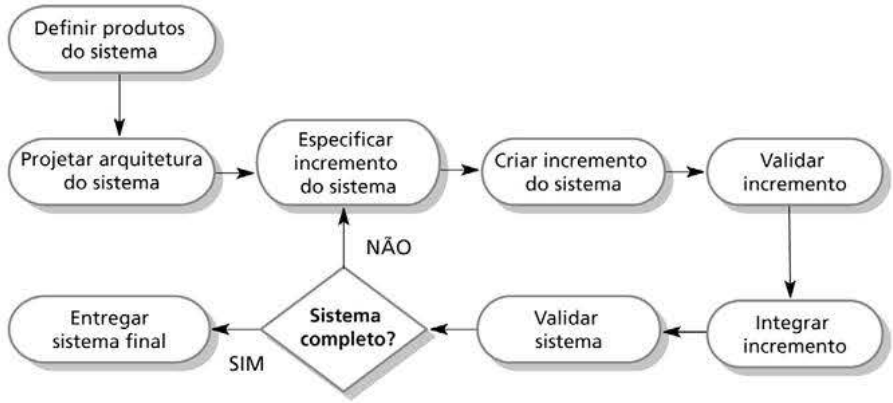
O desenvolvimento de software incremental, na minha visão, é uma abordagem muito melhor para desenvolvimento da maioria dos sistemas de negócios, de *e-commerce* e pessoais, pois ela reflete a maneira fundamental como tendemos a resolver os problemas. Raramente trabalhamos na solução completa de problemas antecipadamente, mas seguimos em direção à solução por meio de série de passos, retrocedendo quando percebemos que cometemos um erro.

Contudo, pode haver dificuldades reais com essas abordagens, particularmente em grandes empresas com procedimentos bastante rígidos, e em organizações nas quais o desenvolvimento de software em geral é terceirizado. As principais dificuldades com o desenvolvimento iterativo e seus incrementos são:

1. *Problemas de gerenciamento*. Estruturas de gerenciamento de software para grandes sistemas são estabelecidas para lidar com um modelo de processo de software que gera produtos regulares para avaliar o progresso. Os sistemas desenvolvidos incrementalmente mudam tão rapidamente que não compensam o custo de produzir grandes quantidades

Figura 17.1

Processo de desenvolvimento iterativo.



de documentação de sistema. Além disso, o desenvolvimento incremental pode, às vezes, requerer tecnologias não conhecidas a serem usadas para assegurar a entrega mais rápida do software. Os gerentes podem achar difícil usar uma equipe existente nos processos de desenvolvimento incremental devido à falta dessas habilidades.

2. *Problemas de contrato*. O modelo de contrato normal entre o cliente e um desenvolvedor de software é baseado em uma especificação de sistema. Quando não há tal especificação, pode ser difícil elaborar um contrato para o desenvolvimento do sistema. Os clientes podem ficar insatisfeitos com um contrato que simplesmente paga os desenvolvedores pelo tempo despendido no projeto, sendo que isso pode levar à lentidão e fazer com que o orçamento seja excedido; os desenvolvedores provavelmente não aceitam um contrato de preço fixo porque eles não podem controlar as alterações solicitadas pelos usuários finais.
3. *Problemas de validação*. Em um processo baseado em especificações, a verificação e a validação são usadas para demonstrar que o sistema atende à sua especificação. Uma equipe independente de V & V pode iniciar o trabalho tão logo a especificação esteja disponível e preparar testes simultâneos com a implementação do sistema. Processos de desenvolvimento iterativos tentam minimizar a documentação e intercalar a especificação e o desenvolvimento. Isso torna difícil a validação independente em sistemas desenvolvidos incrementalmente.
4. *Problemas de manutenção*. As alterações contínuas tendem a corromper a estrutura de qualquer sistema de software. Isso significa que qualquer pessoa além dos desenvolvedores iniciais pode encontrar dificuldades para compreender o software. Uma maneira de reduzir esse problema é usar o refactoring, no qual as estruturas de software são continuamente aprimoradas durante o processo de desenvolvimento. Esse assunto é explicado na Seção 17.2, na qual abordo a *extreme programming*. Além disso, se uma tecnologia especializada, como ambientes RAD (explicados na Seção 17.3), for usada para apoiar o desenvolvimento rápido, a tecnologia RAD pode tornar-se obsoleta. Portanto, encontrar pessoas que tenham o conhecimento necessário para manter o sistema pode ser difícil.

Naturalmente, há alguns tipos de sistemas nos quais o desenvolvimento e a entrega incrementais não são a melhor abordagem. Esses são sistemas muito grandes, nos quais o desenvolvimento pode envolver equipes que trabalham em locais diferentes; alguns sistemas embutidos, nos quais o software depende de desenvolvimento de hardware e alguns sistemas críticos, nos quais todos os requisitos devem ser analisados quanto a interações que possam comprometer a segurança do sistema.

Esses sistemas, naturalmente, sofrem dos mesmos problemas de incerteza e mudança de requisitos. Portanto, para resolver esses problemas e obter algum dos benefícios do desenvolvimento incremental, um processo híbrido pode ser usado no qual um protótipo de sistema é desenvolvido iterativamente e usado como uma plataforma de experimentos dos requisitos de sistema e projeto. Com a experiência obtida com o protótipo, você pode aumentar a confiança de que os requisitos atendem às reais necessidades dos stakeholders do sistema.

Uso o termo *prototipação* no sentido de processo iterativo de desenvolvimento de um sistema experimental que *não* é destinado à disponibilização ao cliente. Um protótipo de sistema é desenvolvido para auxiliar o desenvolvedor de software e o cliente a compreender o que implementar. Entretanto, o termo *prototipação evolucionária* às vezes é usado como um sinônimo para desenvolvimento de software incremental. O protótipo não é descartado, mas sim desenvolvido para atender aos requisitos do cliente.

A Figura 17.2 mostra que o desenvolvimento incremental e a prototipação têm objetivos diferentes:

1. O objetivo do desenvolvimento incremental é entregar um sistema de trabalho aos usuários finais. Isso significa que você deve, geralmente, começar com os requisitos do usuário mais bem compreendidos e com a prioridade mais alta. Requisitos vagos e com baixa prioridade serão implementados quando e se os usuários solicitarem.

Figura 17.2

Desenvolvimento incremental e prototipação.

2. O objetivo da prototipação *throw-away* é validar ou derivar os requisitos de sistema. Você deve começar com requisitos que não são bem compreendidos porque você precisa saber mais sobre eles. Os requisitos simples talvez nunca necessitem ser prototipados.

Outra importante distinção entre essas abordagens está no gerenciamento da qualidade dos sistemas. Protótipos *throw-away* têm um tempo de vida muito curto. Deve ser possível mudá-los rapidamente durante o desenvolvimento, mas a manutenção de longo prazo não é necessária. Um fraco desempenho e baixa confiabilidade podem ser aceitáveis em protótipos *throw-away*, contanto que ele ajude na compreensão dos requisitos.

Por outro lado, sistemas de desenvolvimento incrementais, nos quais versões iniciais evoluem para o sistema final, devem ser desenvolvidos com os mesmos padrões organizacionais de qualidade que qualquer outro software. Eles devem ter uma estrutura robusta que permita mantê-los por muitos anos. Devem ser confiáveis e eficientes e se ajustar aos padrões organizacionais adequados.

17.1 Métodos ágeis

Nas década de 1980 e início da década de 1990, havia uma visão geral de que a melhor maneira de obter o melhor software era por meio de um cuidadoso planejamento de projeto, garantia de qualidade formalizada, uso de métodos de análise e projeto apoiados por ferramentas CASE e controlados por um rigoroso processo de desenvolvimento de software. Essa visão vem, essencialmente, da comunidade de engenharia de software preocupada com sistemas de software grandes e de longa vida, geralmente constituídos de um grande número de programas individuais.

Alguns ou todos esses programas eram freqüentemente sistemas críticos, conforme explicado no Capítulo 3. Esse software era desenvolvido por equipes grandes que algumas vezes trabalhavam para empresas diferentes. Elas estavam, muitas vezes, separadas geograficamente e trabalhavam no software durante longos períodos. Um exemplo desse tipo de software é o sistema de controle de um avião moderno, que pode levar mais de dez anos desde a especificação inicial até sua disponibilização. Essas abordagens, algumas das quais explicadas neste livro, envolvem um overhead significativo de planejamento, projeto e documentação de sistema. Esse overhead é justificado quando o trabalho de várias equipes de desenvolvimento tem de ser coordenado, quando o sistema é crítico e quando muitas pessoas diferentes serão envolvidas na manutenção do software durante seu tempo de vida.

Contudo, quando essa abordagem de desenvolvimento pesada e baseada em planos foi aplicada a sistemas de pequenas e médias empresas, o overhead envolvido foi tão grande que algumas vezes dominou o processo de desenvolvimento de software. O tempo gasto para determinar como o sistema deveria ser desenvolvido era maior do que o empregado no desenvolvimento do programa e em testes. À medida que os requisitos de sistema mudavam, o retrabalho era essencial e, em princípio pelo menos, a especificação e o projeto tinham de mudar com o programa.

A insatisfação com essas abordagens pesadas levou um número de desenvolvedores de software na década de 1990 a propor novos métodos ágeis. Estes permitiam que a equipe de desenvolvimento se concentrasse no software somente, em vez de em seu projeto e documentação. Geralmente, os métodos ágeis contam com uma abordagem iterativa para especificação, desenvolvimento e entrega de software, e foram criados principalmente para apoiar o desenvolvimento de aplicações de negócios nas quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. Eles destinam-se a entregar um software de trabalho rapidamente aos clientes, que podem então propor novos requisitos e alterações a serem incluídos nas iterações posteriores do sistema.

Provavelmente o método ágil mais conhecido é a *extreme programming* (Beck, 1999; Beck, 2000), que descrevo adiante neste capítulo. Contudo, outros métodos ágeis incluem Scrum (Schwaber e Beedle, 2001), Crystal (Cockburn, 2001), Adaptive Software Development (Highsmith, 2000), DSDM (Stapleton, 1997) e Feature Driven Development (Palmer e Felsing, 2002). O sucesso desses métodos levou a alguma integração com métodos de desenvolvimento mais tradicionais baseados em modelagem de sistemas, resultando na noção de modelagem ágil (Ambler e Jeffries, 2002) e em instâncias ágeis do Processo Unificado Racional (Larman, 2002).

Embora esses métodos ágeis sejam todos baseados na noção de desenvolvimento e entrega incrementais, eles propõem processos diferentes para conseguir isso. Contudo, compartilham um conjunto de princípios e, portanto, têm muito em comum. Esses princípios são mostrados na Tabela 17.1.

Tabela 17.1 Princípios dos métodos ágeis

Princípio	Descrição
Envolvimento do cliente	Clientes devem ser profundamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar as iterações do sistema.
Entrega incremental	O software é desenvolvido em incrementos e o cliente especifica os requisitos a serem incluídos em cada incremento.
Pessoas, não processo	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Os membros da equipe devem desenvolver suas próprias maneiras de trabalhar sem processos prescritivos.
Aceite as mudanças	Tenha em mente que os requisitos do sistema vão mudar, por isso projete o sistema para acomodar essas mudanças.
Mantenha a simplicidade	Concentre-se na simplicidade do software que está sendo desenvolvido e do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Os adeptos dos métodos ágeis exageraram ao promover seu uso, chegando a encobrir suas imperfeições. Isso gerou uma resposta igualmente extrema, que, na minha visão, superestima os problemas com essa abordagem (Stephens e Rosenberg, 2003). Críticos mais razoáveis tais como DeMarco e Boehm (DeMarco e Boehm, 2002) destacam tanto as vantagens como as desvantagens dos métodos ágeis. Eles defendem que uma abordagem híbrida na qual os métodos ágeis incorporam algumas técnicas de desenvolvimento baseado em planos pode ser a melhor solução.

Na prática, contudo, os princípios básicos dos métodos ágeis são, às vezes, difíceis de compreender:

1. Enquanto a idéia de envolvimento do cliente no processo de desenvolvimento é atrativa, seu sucesso depende de um cliente disposto e capaz de despender tempo com a equipe de desenvolvimento e que possa representar todos os stakeholders do sistema. Freqüentemente, os representantes do cliente estão sujeitos a outras pressões e não podem participar totalmente do desenvolvimento do software.
2. Membros individuais da equipe podem não ter a personalidade adequada para o envolvimento intenso típico dos métodos ágeis. Eles podem não interagir bem com outros membros da equipe.
3. A priorização de mudanças pode ser extremamente difícil, especialmente em sistemas em que há muitos stakeholders. Em geral, cada stakeholder adota diferentes prioridades para diferentes mudanças.
4. Manter a simplicidade requer trabalho extra. Sob a pressão dos cronogramas de entrega, os membros da equipe podem não ter tempo de realizar as simplificações desejáveis no sistema.

Outro problema não técnico, que é um problema geral no desenvolvimento e entrega incrementais, ocorre quando o cliente usa uma organização externa para o desenvolvimento do sistema. Conforme expliquei no Capítulo 6, a lista de requisitos de software é geralmente parte do contrato entre cliente e fornecedor. Devido à especificação incremental ser inerente aos métodos ágeis, definir o contrato para esse tipo de desenvolvimento pode ser difícil.

Conseqüentemente, os métodos ágeis têm de contar com contratos em que o cliente paga pelo tempo necessário para o desenvolvimento do sistema, em vez de pelo desenvolvimento de um conjunto específico de requisitos. Desde que tudo esteja bem, isso beneficia tanto o cliente quanto o desenvolvedor. Contudo, se surgirem problemas pode haver disputas difíceis sobre quem tem a culpa e quem deve pagar pelo tempo extra e recursos necessários para resolver os problemas.

Todos os métodos têm limites, e os métodos ágeis são somente adequados para alguns tipos de desenvolvimento de sistema. Na minha opinião, eles são mais adequados para o desenvolvimento de sistemas de pequenas e médias empresas e produtos para computadores pessoais. Eles não são tão adequados para desenvolvimento de sistemas de larga escala com as equipes de desenvolvimento em lugares diferentes e onde possa haver interações complexas com outros sistemas de hardware e software. Os métodos ágeis também não devem ser usados para desenvolvimento de sistemas críticos nos quais é necessária uma análise detalhada de todos os requisitos de sistema para compreender suas implicações de segurança.

17.2 *Extreme programming*

A *extreme programming* (XP) é talvez o mais conhecido e mais amplamente usado dos métodos ágeis. O nome foi cunhado por Beck (Beck, 2000) porque a abordagem foi desenvolvida pelo avanço de reconhecida boa prática, tal como o desenvolvimento iterativo e o envolvimento do cliente em níveis 'extremos'.

Na *extreme programming*, todos os requisitos são expressos como cenários (chamados histórias do usuário), que são implementados diretamente como uma série de tarefas. Os programadores trabalham em pares e desenvolvem testes para cada tarefa antes da escrita do código. Todos os testes devem ser executados com sucesso quando um novo código é integrado ao sistema. Há um pequeno espaço de tempo entre os releases do sistema. A Figura 17.3 ilustra o processo XP para produzir um incremento do sistema que está sendo desenvolvido.

Extreme programming envolve um número de práticas, resumidas na Tabela 17.2, que se enquadram nos princípios dos métodos ágeis:

1. O desenvolvimento incremental é apoiado por pequenos e freqüentes releases do sistema e por uma abordagem de descrição de requisitos baseada nas histórias ou cenários do cliente que podem ser a base para o planejamento do processo.
2. O envolvimento do cliente é apoiado pelo engajamento em tempo integral deste na equipe de desenvolvimento. O representante do cliente faz parte do desenvolvimento e é responsável pela definição de testes de aceitação do sistema.

Figura 17.3

Ciclo de um release em *extreme programming*

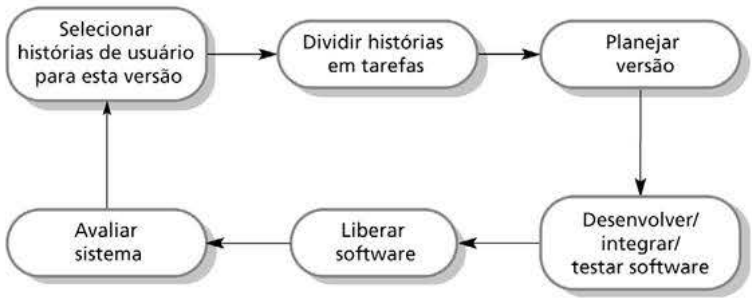


Tabela 17.2 Práticas da *extreme programming*

Princípio ou prática	Descrição
Planejamento incremental	Os requisitos são registrados em cartões de histórias e as histórias a serem incluídas em um release são determinadas pelo tempo disponível e sua prioridade relativa. Os desenvolvedores dividem essas histórias em 'tarefas'. Veja as figuras 17.4 e 17.5.
Pequenos releases	O conjunto mínimo útil de funcionalidade que agrega valor ao negócio é desenvolvido primeiro. Releases do sistema são freqüentes e adicionam funcionalidade incrementalmente ao primeiro release.
Projeto simples	É realizado um projeto suficiente para atender aos requisitos atuais e nada mais.
Desenvolvimento *test-first*	Um framework automatizado de teste unitário é usado para escrever os testes para uma nova parte da funcionalidade antes que esta seja implementada.
Refactoring	Espera-se que todos os desenvolvedores recriem o código continuamente tão logo os aprimoramentos do código forem encontrados. Isso torna o código simples e fácil de manter.
Programação em pares	Os desenvolvedores trabalham em pares, um verificando o trabalho do outro e fornecendo apoio para realizar sempre um bom trabalho.
Propriedade coletiva	Os pares de desenvolvedores trabalham em todas as áreas do sistema, de tal maneira que não se formem ilhas de conhecimento, com todos os desenvolvedores de posse de todo o código. Qualquer um pode mudar qualquer coisa.
Integração contínua	Tão logo o trabalho em uma tarefa seja concluído, este é integrado ao sistema como um todo. Depois de qualquer integração, todos os testes unitários do sistema devem ser realizados.
Ritmo sustentável	Grandes quantidades de horas extras não são consideradas aceitáveis, pois, no médio prazo, há uma redução na qualidade do código e na produtividade.
Cliente *on-site*	Um representante do usuário final do sistema (o cliente) deve estar disponível em tempo integral para apoiar a equipe de XP. No processo da *extreme programming*, o cliente é um membro da equipe de desenvolvimento e é responsável por trazer os requisitos do sistema à equipe para implementação.

3. As pessoas, não o processo, são apoiadas ao dividirem a programação com seus pares, partilhando a propriedade do código do sistema e um processo de desenvolvimento sustentável que não envolve excessivas horas de trabalho.
4. As mudanças são apoiadas por meio de releases regulares de sistema, desenvolvimento *test-first* e integração contínua.
5. A manutenção da simplicidade é apoiada pelo refactoring constante para aprimorar a qualidade do código e o uso de projetos simples que não antecipam mudanças futuras no sistema.

Em um processo XP, os clientes estão intimamente envolvidos na especificação e priorização dos requisitos de sistema. Os requisitos não são especificados como listas de funções necessárias do sistema. Em vez disso, o cliente é parte da equipe de desenvolvimento e discute cenários com os outros membros da equipe. Juntos, eles desenvolvem um 'cartão de histórias' que engloba as necessidades do cliente. A equipe de desenvolvimento, então, volta-se para a implementação daquele cenário em um release futuro do software. Um exemplo de um cartão de histórias para o sistema LIBSYS, baseado em um cenário descrito no Capítulo 6, é mostrado na Figura 17.4.

Após o desenvolvimento dos cartões de histórias, a equipe de desenvolvimento os dividirá em tarefas e estimará o esforço e os recursos necessários para implementação. O cliente então prioriza as histórias para implementação, escolhendo as que podem ser usadas imediatamente para proporcionar apoio útil ao negócio. Naturalmente, quando os requisitos mudam, as histórias que não foram implementadas mudam ou são descartadas. Se forem necessárias mudanças em um sistema que já foi entregue, novos cartões de histórias são desenvolvidos e, novamente, o cliente decide se essas mudanças devem ter prioridade sobre a nova funcionalidade.

A *extreme programming* exige uma abordagem 'extrema' para o desenvolvimento iterativo. Novas versões de software podem ser compiladas várias vezes por dia e os incrementos são entregues para os clientes aproximadamente a cada duas semanas. Quando um programador compila o sistema para criar uma nova versão, ele deve executar todos os testes automatizados existentes bem como os testes para a nova funcionalidade. A nova compilação do software será aceita somente se todos os testes foram executados com sucesso.

Um preceito fundamental da engenharia de software tradicional é que você deve projetar para a mudança. Ou seja, você deve antecipar mudanças futuras para o software e projetá-lo de tal maneira que essas mudanças possam ser implementadas facilmente. A *extreme programming*, contudo, descarta esse princípio alegando que projetar para a mudança é, geralmente, um esforço inútil. As mudanças antecipadas muitas vezes não ocorrem e as solicitações de mudança realizadas são completamente diferentes.

O problema com a implementação de mudanças não antecipadas é que elas tendem a degradar a estrutura do software, fazendo com que as mudanças tornem-se cada vez mais difíceis de implementar. A *extreme programming* lida com este problema defendendo que o software deve passar por refactoring constantemente. Isso significa que a equipe de programação procura por possíveis melhorias no software, implementando-as imediatamente. Portanto, o software deve sempre ser fácil de compreender e alterar quando novas histórias são implementadas.

17.2.1 Teste em XP

Conforme expliquei na introdução deste capítulo, uma das importantes diferenças entre o desenvolvimento iterativo e o desenvolvimento baseado em planos é a maneira como o sistema é testado. Com o desenvolvimento iterativo, não há uma especificação de sistema que possa ser usada por uma equipe de teste externa para desenvolver os testes de sistema. Como conseqüência, algumas abordagens para o desenvolvimento iterativo têm um processo de teste muito informal.

Figura 17.4

Cartão de histórias para baixar documentos.

Baixando e imprimindo um artigo
Primeiro, você seleciona o artigo que deseja em uma lista. Depois você precisa informar ao sistema como quer pagar pelo artigo — isso pode ser feito por meio de uma assinatura, de uma conta empresarial ou por cartão de crédito. Após, você obtém do sistema um formulário de direitos autorais para preenchimento. Após enviar o formulário, o artigo desejado é baixado para seu computador. Em seguida, você escolhe uma impressora para imprimir uma cópia do artigo. Você informa ao sistema que a impressão foi bem-sucedida. Se o artigo for somente para impressão, você não poderá manter uma versão em PDF, de modo que o artigo será excluído automaticamente de seu computador.

Para evitar alguns dos problemas de teste e validação de sistema, a *extreme programming* (XP) enfatiza mais o processo de teste que outros métodos ágeis. O teste de sistema é fundamental para a XP, segundo a qual uma abordagem desenvolvida para reduzir a probabilidade de produção de novos incrementos de sistemas introduzirá erros no software existente.

As características principais de teste na XP são:

1. Desenvolvimento *test-first*
2. Desenvolvimento incremental de teste a partir de cenários
3. Envolvimento do usuário no desenvolvimento e validação de testes
4. O uso de ferramentas de teste automatizadas

O desenvolvimento *test-first* é uma das mais importantes inovações na XP. A escrita de testes primeiro define implicitamente tanto uma interface como uma especificação do comportamento para a funcionalidade que está sendo desenvolvida. Os problemas de mau entendimento de requisitos e interface são reduzidos. Essa abordagem pode ser adotada em qualquer processo em que houver uma clara relação entre um requisito de sistema e a implementação de código necessária. Na XP, você pode sempre ver essa ligação porque os cartões de histórias que representam os requisitos são divididos em tarefas, e as tarefas são a unidade principal de implementação.

Conforme expliquei, os requisitos de usuário na XP são expressos como cenários ou histórias, e o usuário prioriza esses requisitos para o desenvolvimento. A equipe de desenvolvimento avalia cada cenário e o divide em tarefas. Cada tarefa representa uma característica discreta do sistema e um teste unitário pode então ser projetado para essa tarefa. Por exemplo, alguns dos cartões de tarefas desenvolvidos a partir do cartão de histórias para baixar documentos (Figura 17.4) são mostrados na Figura 17.5.

Cada tarefa gera um ou mais testes unitários que verificam a implementação descrita na tarefa. Por exemplo, a Figura 17.6 é uma descrição resumida de um caso de teste desenvolvido para verificar se um número de cartão de crédito válido foi implementado.

O papel do cliente no processo de teste é auxiliar no desenvolvimento de testes de aceitação para as histórias que deverão ser implementadas no próximo release do sistema. Conforme explicado no Capítulo 23, o teste de aceitação é o processo no qual o sistema é testado usando-se dados do cliente para verificar se ele atende às reais necessidades do cliente. Em XP, o teste de aceitação, assim como o desenvolvimento, é incremental. Para essa história em particular, o teste de aceitação envolve a realização de várias seleções de documentos, pagar por eles de maneiras diferentes e imprimi-los em impressoras diferentes. Na prática, provavelmente seria desenvolvida uma série de testes de aceitação em vez de um único teste.

O desenvolvimento *test-first* e o uso de ferramentas de teste automatizadas são os principais pontos fortes da abordagem XP. Em vez de escrever o código do programa e depois os testes desse código, o desenvolvimento *test-first* propõe que o teste seja escrito antes do código. O teste é escrito como um componente executável antes de a tarefa ser implementada. Após a implementação do software, o teste pode ser executado imediatamente. Esse teste de componente deve ser independente, deve simular o envio da entrada a ser testada e verificar se o resultado atende à especificação de saída. A ferramenta de teste automatizada é um sistema que envia esses testes automatizados para execução.

Com o desenvolvimento *test-first*, há sempre um conjunto de testes que podem ser rápida e facilmente executados. Isso significa que, quando qualquer funcionalidade for adicionada ao sistema, os testes poderão ser executados e os problemas que o novo código introduziu poderão ser imediatamente detectados.

Figura 17.5

Cartões de tarefa para baixar documentos.

Tarefa 1: Implementar fluxo de trabalho principal

Tarefa 2: Implementar catálogo e seleção de artigos

Tarefa 3: Implementar arrecadação de pagamentos

O pagamento pode ser feito de três maneiras diferentes. O usuário seleciona por meio de qual maneira deseja pagar. Se o usuário tiver uma assinatura de biblioteca, ele poderá inserir a chave de assinante que deve ser verificada pelo sistema. Ele também pode inserir um número de conta da empresa. Caso este seja válido, um débito do custo do artigo será enviado para essa conta. Finalmente, ele poderá inserir um número de cartão de crédito de 16 dígitos e data de expiração para validação e, caso sejam válidos, um débito será enviado para a conta do cartão de crédito.

Figura 17.6

Descrição de caso de teste para validação de cartão de crédito.

Baixando e imprimindo um artigo
Entrada: Uma string que representa o número de cartão de crédito e dois inteiros que representam o mês e o ano de expiração do cartão.
Testes: Verificar se todos os bytes na string são dígitos. Verificar se o mês varia entre 1 e 12 e o ano é posterior ou igual ao ano atual. Usando os primeiros 4 dígitos do número de cartão de crédito, verificar se o emissor do cartão é válido consultando a tabela de emissores de cartões. Verificar a validade do cartão de crédito enviando o número do cartão e a data de expiração para o emissor de cartões.
Saída: OK ou mensagem de erro indicando que o cartão é inválido.

No desenvolvimento *test-first*, os implementadores de tarefa têm de compreender detalhadamente a especificação de modo que possam escrever o teste para o sistema. Isso significa que ambigüidades e omissões na especificação têm de ser esclarecidas antes do início da implementação. Além disso, é evitado o problema de 'test-lag', no qual o desenvolvedor do sistema trabalha num ritmo mais rápido que o testador, fazendo com que a implementação caminhe muito além do teste, criando a tendência de pular testes de modo a manter o cronograma.

Contudo, o desenvolvimento *test-first* nem sempre funciona como previsto. Os programadores preferem programar e testar e, algumas vezes, escrever testes incompletos que não verificam situações excepcionais. Além disso, alguns testes podem ser muito difíceis de escrever. Por exemplo, numa interface de usuário complexa, freqüentemente é difícil escrever testes unitários para o código que implementa a lógica de exibição e o fluxo de trabalho entre telas. Finalmente, é difícil julgar a abrangência de um conjunto de testes. Embora você possa ter uma grande quantidade de testes de sistema, seu conjunto de testes pode não fornecer uma cobertura completa. Partes cruciais do sistema podem não ser executadas e permanecerem assim sem terem sido testadas.

Contar com o cliente para apoiar o desenvolvimento de testes de aceitação é, algumas vezes, a principal dificuldade no processo de teste da XP. As pessoas no papel de cliente têm tempo disponível muito limitado e podem não ser capazes de trabalhar em tempo integral com a equipe de desenvolvimento. O cliente pode sentir que fornecer os requisitos foi suficiente e relutar em se envolver no processo de teste.

17.2.2 Programação em pares

Outra prática inovadora consiste em os programadores trabalharem em pares para desenvolver o software. Eles realmente sentam-se juntos em frente da mesma workstation para desenvolver o software. O desenvolvimento não envolve sempre o mesmo par de pessoas que trabalham juntos. Em vez disso, a idéia é que os pares sejam criados dinamicamente de tal modo que todos os membros da equipe possam trabalhar com outros membros numa programação em pares durante o processo de desenvolvimento.

O uso de programação em pares tem várias vantagens:

1. Apóia a idéia de propriedade e responsabilidade comuns para o software. Isso reflete a idéia de Weinberg sobre programação sem egoísmo (Weinberg, 1971), na qual o software é de propriedade da equipe como um todo e as pessoas não são responsabilizadas por problemas com o código. Em vez disso, a equipe tem responsabilidade coletiva na resolução desses problemas.
2. Atua como um processo informal de revisão porque cada linha de código é vista por pelo menos duas pessoas. Inspeções de código e revisões (abordadas no Capítulo 22) são muito bem-sucedidas em descobrir uma alta porcentagem de erros de software. No entanto, isso consome tempo para organização e, geralmente, causa atrasos no processo de desenvolvimento. Embora a programação em pares seja um processo menos formal que provavelmente não encontre muitos erros, é um processo de inspeção mais barato que inspeções formais de programa.
3. Auxilia no apoio ao refactoring, que é um processo de aprimoramento de software. Um princípio da XP é que o software deve ser constantemente reconstruído. Isto é, partes do código devem ser reescritas para aprimorar sua clareza ou estrutura. A dificuldade de implementação em um ambiente normal de desenvolvimento é o fato de isso

ser um esforço que obtém benefício no longo prazo, e uma pessoa que pratica o refactoring pode ser considerada menos eficiente que outra que simplesmente realiza o desenvolvimento de código. Onde a programação em pares e a propriedade coletiva são usadas, outras pessoas ganham imediatamente com o refactoring, de forma que elas provavelmente apoiarão o processo.

Você pode pensar que a programação em pares é menos eficiente que a programação individual e que um par de desenvolvedores produz metade da quantidade de código de duas pessoas que trabalham sozinhas. Estudos de desenvolvimentos com XP, contudo, não confirmam isso. A produtividade de desenvolvimento com a programação em pares parece ser comparável a duas pessoas que trabalham independentemente (Williams, et al., 2000). As razões para isso são que os pares discutem o software antes do desenvolvimento e, assim, provavelmente, têm poucos inícios problemáticos e menos retrabalho, e o número de erros evitados pela inspeção informal é tal que menos tempo é empregado na correção de erros descobertos durante o processo de teste.

17.3 Desenvolvimento rápido de aplicações

Embora os métodos ágeis como uma abordagem de desenvolvimento iterativa tenham recebido uma grande atenção nos últimos anos, os sistemas de negócios foram desenvolvidos iterativamente por muitos anos usando técnicas rápidas de desenvolvimento de aplicações. Técnicas rápidas de desenvolvimento de aplicações (RAD – Rapid Application Development Techniques) desenvolveram-se das então chamadas linguagens de quarta geração na década de 1980 e são usadas para o desenvolvimento de aplicações orientadas a dados. Conseqüentemente, elas são geralmente organizadas como um conjunto de ferramentas que permitem que dados sejam criados, pesquisados, exibidos e apresentados em relatórios. A Figura 17.7 ilustra a organização típica de um sistema RAD.

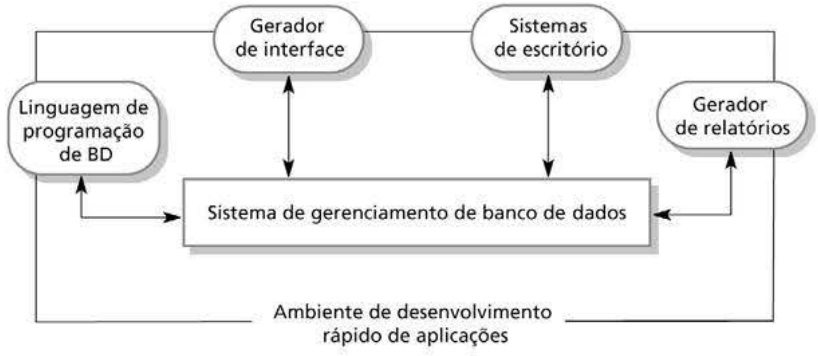
As ferramentas incluídas em um ambiente RAD são:

1. *Uma linguagem de programação de banco de dados* que incorpora conhecimento de estruturas de banco de dados e inclui operações fundamentais de manipulação de banco de dados. SQL (Groff, et al., 2002) é uma linguagem padrão de programação de banco de dados. Os comandos SQL podem ser inseridos diretamente ou gerados automaticamente com base em formulários preenchidos por um usuário final.
2. *Um gerador de interface* que é usado para criar formulários para entrada e exibição de dados.
3. *Links* para aplicações de escritório tais como uma planilha para análise e manipulação de informações numéricas ou um processador de texto para a criação de templates.
4. *Um gerador de relatórios* usado para definir e criar relatórios baseado em informações de um banco de dados. Os sistemas RAD são bem-sucedidos, pois, como expliquei no Capítulo 13, há uma grande quantidade de aplicações comuns de negócios. Em essência, essas aplicações estão freqüentemente voltadas à atualização de banco de dados e produção de relatórios com base em informações do banco de dados. Os formulários-padrão são usados para entrada e saída. Os sistemas RAD são dirigidos para a produção de aplicações interativas que contam com a abstração de informações de um banco de dados organizacional, apresentação para usuários finais em seus terminais ou estações de trabalho e atualização do banco de dados com as mudanças realizadas pelos usuários.

Muitas aplicações de negócios contam com formulários estruturados para entrada e saída, assim os ambientes RAD fornecem recursos poderosos para definição de telas e geração de relatórios. As telas são freqüentemente definidas como uma série de formulários ligados (em uma aplicação que estudamos, havia 137 definições de formulários).

Figura 17.7

Ambiente de desenvolvimento rápido de aplicações.



Sendo assim um sistema de geração de telas deve fornecer:

1. *Definição de formulários interativos* nos quais o desenvolvedor define os campos para serem exibidos e como estes devem ser organizados.
2. *Ligação de formulários* em que o desenvolvedor pode especificar quais entradas específicas fazem com que formulários adicionais sejam exibidos.
3. *Verificação de campos* em que o desenvolvedor define intervalos permitidos de entrada de valores para os campos de formulário.

Todos os ambientes RAD atualmente apóiam o desenvolvimento de interfaces de banco de dados baseados em navegadores Web. Eles permitem que o banco de dados seja acessado de qualquer lugar através de uma conexão com a Internet válida. Isso reduz os custos de treinamento e software e permite que usuários externos tenham acesso ao banco de dados. Contudo, as limitações inerentes dos navegadores Web e protocolos de Internet significam que essa abordagem pode ser inadequada para sistemas em que são exigidas respostas rápidas e interativas.

Muitos sistemas RAD atuais também incluem ferramentas de programação visual que permitem que o sistema seja desenvolvido interativamente. Em vez de escrever um programa seqüencial, o desenvolvedor de sistema manipula ícones gráficos representando funções, dados e componentes de interface com o usuário, e associa scripts de processamento a esses ícones. Um programa executável é gerado automaticamente a partir da representação visual do sistema.

Os sistemas de desenvolvimento visual, tais como o Visual Basic, apóiam essa abordagem de desenvolvimento de aplicações. Os programadores de aplicações criam o sistema interativamente pela definição da interface em termos de telas, campos, botões e menus. Estes são definidos e scripts de processamento são associados a partes individuais da interface (por exemplo, um botão chamado Simular). Esses scripts podem ser chamados para componentes reutilizáveis, código com finalidade especial ou uma combinação de ambos.

Essa abordagem é ilustrada na Figura 17.8, que mostra uma tela de aplicação incluindo menus na parte superior, campos de entrada (os campos brancos à esquerda), campos de saída (o campo cinza à esquerda) e botões (os retângulos com cantos arredondados à direita). Quando esses elementos são posicionadas no display pelo sistema de programação visual, o desenvolvedor define qual componente reutilizável deve ser associado a ele ou escreve um trecho de programa para realizar o processamento necessário. A Figura 17.8 também mostra os componentes associados a alguns dos elementos do display.

O Visual Basic é um exemplo muito sofisticado para uma linguagem de scripting (Ousterhout, 1998). As linguagens de scripting são linguagens sem tipo e de alto nível projetadas para ajudá-lo a integrar componentes e criar sistemas. Um antigo exemplo de uma linguagem de scripting foi o shell Unix (Bourne, 1978; Gordon e Bieman, 1995); desde seu desenvolvimento, várias linguagens de scripting mais poderosas foram criadas (Ousterhout, 1994; Lutz, 1996; Wall, et al., 1996). As linguagens de scripting incluem estruturas de controle e conjunto de ferramentas gráficas que, como Ousterhout (Ousterhout, 1998) ilustra, podem reduzir significativamente o tempo necessário para o desenvolvimento do sistema.

Essa abordagem para desenvolvimento de sistema permite que o desenvolvimento rápido de aplicações relativamente simples possa ser realizado por uma equipe pequena de pessoas. Para sistemas maiores que devem ser desenvolvidos por

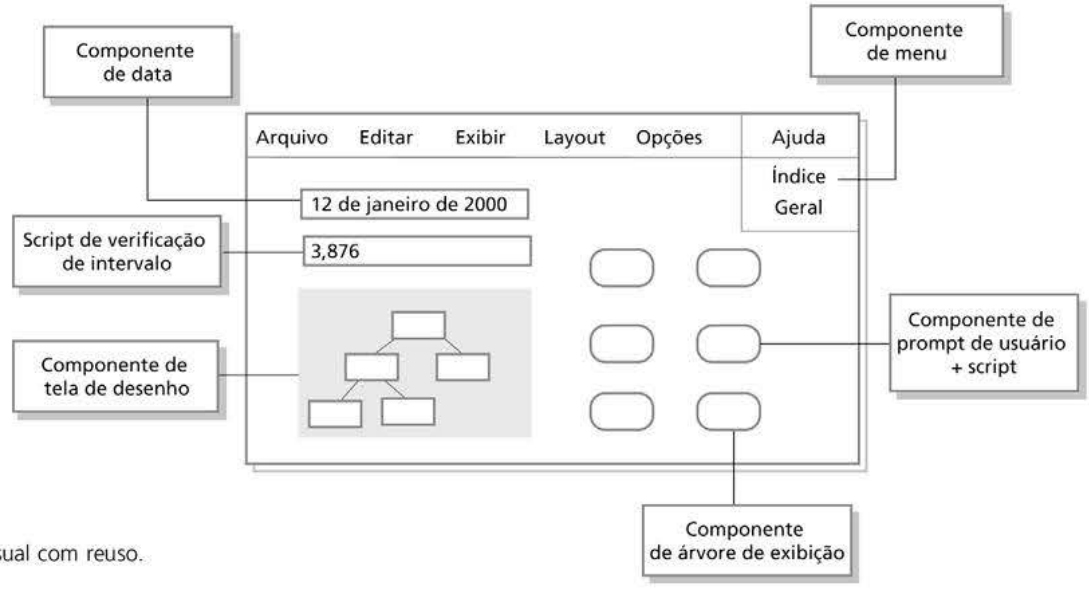
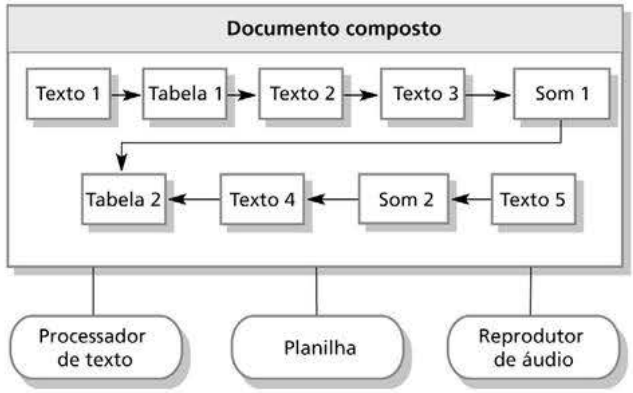


Figura 17.8

Programação visual com reuso.

Figura 17.9

Relacionamento entre as aplicações.



equipes também maiores, essa abordagem torna-se mais difícil de organizar. Não há uma arquitetura de sistema explícita e há, muitas vezes, dependências complexas entre partes do sistema que podem causar dificuldades quando mudanças são solicitadas. Além disso, devido às linguagens de scripting serem limitadas a um conjunto específico de objetos de interação, a implementação de interfaces com o usuário não padronizadas pode ser difícil.

O *desenvolvimento visual* é uma abordagem para RAD que conta com a integração de componentes de software muito pequenos e reutilizáveis. Uma abordagem alternativa baseada em reuso reaproveita 'componentes' que são sistemas completos de aplicação. Isso é chamado algumas vezes de desenvolvimento baseado em COTS (Commercial Off-The-Shelf), ou seja, 'de prateleira' — as aplicações já estão disponíveis. Por exemplo, se um sistema requer um recurso de processamento de texto, você pode usar um sistema padrão de processamento de palavras tal como o Microsoft Word. Eu explico o desenvolvimento baseado em COTS de uma perspectiva de reuso no Capítulo 18.

Para ilustrar o tipo de aplicação que poderia ser desenvolvido usando uma abordagem baseada em COTS, considere o processo de gerenciamento de requisitos explicado no Capítulo 7. Um sistema de apoio ao gerenciamento de requisitos precisa de uma maneira de captar os requisitos, armazená-los, produzir relatórios, descobrir relacionamentos entre os requisitos e gerenciar esses relacionamentos como tabelas de rastreabilidade. Em uma abordagem baseada em COTS, um protótipo poderia ser criado pela ligação de uma base de dados (para armazenar requisitos), um processador de texto (para captar requisitos e formatar relatórios), uma planilha (para gerenciar tabelas de rastreabilidade) e códigos especialmente escritos para localizar relacionamentos entre os requisitos.

O desenvolvimento baseado em COTS dá ao desenvolvedor acesso a todas as funcionalidades de uma aplicação. Se a aplicação também fornece recursos de scripting ou tailoring (por exemplo, macros do Excel), eles podem ser usados para desenvolver alguma funcionalidade da aplicação. Uma metáfora de documentos compostos é útil para compreender essa abordagem de desenvolvimento de aplicações. Os dados processados pelo sistema podem ser organizados em um documento composto que atua como um contêiner de vários objetos. Esses objetos contêm tipos diferentes de dados (tais como uma tabela, um diagrama, um formulário) que podem ser processados por aplicações diferentes. Os objetos são ligados e classificados de tal maneira que o acesso a um objeto resulta na inicialização da aplicação associada.

A Figura 17.9 ilustra um sistema de aplicação constituído de um documento composto que inclui elementos de texto, elementos de planilha e arquivos de som. Os elementos de texto são processados pelo processador de texto, as tabelas pela aplicação de planilha e os arquivos de som pelo reprodutor de áudio. Quando um usuário de sistema acessa um objeto de determinado tipo, a aplicação associada é chamada para prover a funcionalidade específica ao usuário. Por exemplo, quando objetos do tipo som são acessados, o reprodutor de áudio é chamado para processá-los.

A principal vantagem dessa abordagem é que uma grande quantidade de funcionalidade de aplicação pode ser implementada rapidamente a um custo muito baixo. Os usuários que já estão familiarizados com as aplicações que compõem o sistema não têm de aprender como usar novos recursos. Contudo, se eles não sabem como usar as aplicações, o aprendizado pode ser difícil, especialmente se eles forem confundidos pela funcionalidade de aplicação que não é necessária. Pode haver também problemas de desempenho com a aplicação por causa da necessidade de alternar de uma aplicação para outra. O excesso de alternância depende do suporte ao sistema operacional fornecido.

17.4 Prototipação de software

Como expliquei na introdução deste capítulo, há algumas circunstâncias em que, por razões práticas ou contratuais, um processo incremental de entrega de software não pode ser usado. Nessas situações, uma declaração de requisitos de sistema é preenchida e utilizada pela equipe de desenvolvimento como base para o sistema de software. Conforme

expliquei, você pode obter algum dos benefícios do processo de desenvolvimento incremental pela criação de um protótipo de software. Essa abordagem é, algumas vezes, chamada de prototipação *throw-away* porque o protótipo não é entregue para o cliente ou mantido pelo desenvolvedor.

Um protótipo é uma versão inicial de um sistema de software usado para demonstrar conceitos, experimentar opções de projeto e, geralmente, conhecer mais sobre o problema e suas possíveis soluções. Desenvolvimento rápido e iterativo do protótipo é essencial, de modo que os custos são controlados e os stakeholders do sistema podem experimentar o protótipo mais cedo no processo de software.

Um protótipo de software pode ser usado em um processo de desenvolvimento de software de várias maneiras:

1. No processo de engenharia de requisitos, um protótipo pode ajudar na descoberta e validação dos requisitos do sistema.
2. No processo de projeto do sistema, um protótipo pode ser usado para explorar soluções específicas de software e apoiar o projeto de interface com o usuário.
3. No processo de teste, um protótipo pode ser usado para realizar testes completos com o sistema que será entregue para o cliente.

Protótipos de sistema permitem que os usuários experimentem como o sistema apóia seu trabalho. Eles podem ter novas idéias para os requisitos e encontrar pontos fortes e fracos no software. Eles podem, então, propor novos requisitos de sistema. Além disso, quando o protótipo é desenvolvido, pode revelar erros e omissões nos requisitos propostos. Uma função descrita numa especificação pode parecer útil e bem definida. Contudo, quando essa função é combinada com outras funções, os usuários muitas vezes descobrem que sua visão inicial foi incorreta ou incompleta. A especificação de sistema pode, então, ser modificada para refletir a mudança no entendimento dos requisitos.

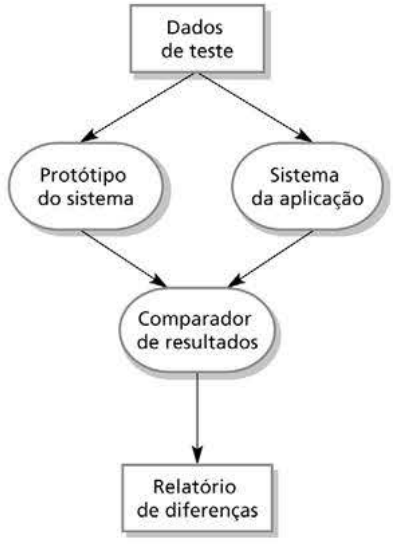
Um protótipo de sistema pode ser usado enquanto o sistema está sendo projetado para realizar experimentos e verificar a viabilidade de um projeto proposto. Por exemplo, um projeto de um banco de dados pode ser prototipado e testado para verificar se permite acesso de dados mais eficiente para as consultas de usuário mais comuns. A prototipação é também uma parte essencial do processo de projeto da interface com o usuário. Devido à natureza dinâmica das interfaces de usuário, descrições textuais e diagramas não são bons o suficiente para expressar os requisitos de interface com o usuário. Portanto, a prototipação rápida com envolvimento do usuário final é a maneira mais lógica para desenvolver interfaces gráficas com o usuário para sistemas de software.

Um grande problema no teste de sistema é a validação, na qual você deve verificar se os resultados de um teste são os esperados. Quando um protótipo de sistema está disponível, você pode reduzir o esforço envolvido na verificação de resultados pela realização de testes back-to-back (Figura 17.10). Os mesmos casos de teste são submetidos ao protótipo e ao sistema em teste. Se ambos os sistemas produzirem o mesmo resultado, provavelmente o caso de teste não detectou falhas. Se os resultados diferirem, pode significar que há uma falha de sistema e as razões da diferença devem ser investigadas.

Finalmente, assim como as atividades de apoio ao processo de software, protótipos podem ser usados para reduzir o tempo necessário para desenvolver a documentação do usuário e treinar os usuários com o sistema. Um sistema de trabalho, embora limitado, fica disponível rapidamente para demonstrar a viabilidade e utilidade da aplicação para a gerência.

Figura 17.10

Teste back-to-back.



Em um estudo de 39 projetos de prototipação, Gordon e Bieman (Gordon e Bieman, 1995) identificaram que os benefícios do uso da prototipação eram:

1. Usabilidade aprimorada do sistema
2. Adequação maior do sistema às necessidades do usuário
3. Qualidade do projeto aprimorada
4. Facilidade de manutenção aprimorada
5. Esforço de desenvolvimento reduzido

O estudo deles sugere que os aprimoramentos na usabilidade e melhores requisitos do usuário que se originam do uso de um protótipo não significam necessariamente um aumento global nos custos de desenvolvimento do sistema. A prototipação geralmente aumenta os custos nos estágios iniciais do processo do software, mas reduz esses custos nos estágios mais avançados do processo de desenvolvimento. A razão principal para isso é que o retrabalho durante o desenvolvimento é evitado devido às poucas mudanças de sistema solicitadas pelos clientes. Contudo, Gordon e Bieman descobriram que o desempenho global do sistema às vezes é degradado se códigos ineficientes de protótipo forem reutilizados.

Um modelo de processo para desenvolvimento de protótipo é mostrado na Figura 17.11. Os objetivos da prototipação devem estar explícitos no início do processo. Eles podem ser o desenvolvimento de um sistema para prototipar a interface com o usuário, desenvolver um sistema para validar os requisitos funcionais de sistema ou desenvolver um sistema para demonstrar a viabilidade da aplicação para a gerência. O mesmo protótipo não pode atender a todos os objetivos. Se os objetivos não forem estabelecidos, a gerência ou os usuários finais podem não compreender a função do protótipo. Conseqüentemente, eles podem não obter os benefícios esperados do desenvolvimento do protótipo.

O próximo estágio no processo é decidir o que será considerado e, talvez o mais importante, o que não fará parte do sistema de protótipo. Para reduzir os custos de prototipação e acelerar o cronograma de entrega, você pode deixar alguma funcionalidade fora do protótipo. Você pode decidir dar menor importância aos requisitos não funcionais tais como tempo de resposta e uso de memória. O tratamento e gerenciamento de erros podem ser ignorados ou podem ser rudimentares já que o objetivo do protótipo é definir uma interface com o usuário. Padrões de confiabilidade e qualidade do programa podem ser reduzidos.

O estágio final do processo é a avaliação do protótipo. Recursos devem ser fornecidos durante esse estágio para o treinamento de usuários, e os objetivos do protótipo devem ser usados para derivar um plano de avaliação. Os usuários precisam de tempo para se adaptarem ao novo sistema e atingir um padrão normal de uso. Uma vez que eles estejam usando o sistema normalmente, serão descobertos erros e omissões de requisitos.

Um problema geral com o desenvolvimento de um protótipo *throw-away* executável é que o modo de uso do protótipo pode não corresponder a como o sistema final entregue será usado. O testador do protótipo pode não ser um típico usuário do sistema. O tempo de treinamento durante a avaliação do protótipo pode ser insuficiente. Se o protótipo é lento, os avaliadores podem ajustar seus modos de operação e evitar que esses recursos do sistema tenham tempos de resposta lentos. Quando fornecido com resposta melhor no sistema final, eles podem usá-lo de maneira diferente.

Os desenvolvedores às vezes são pressionados pelos gerentes a entregarem protótipos *throw-away*, particularmente quando há atrasos na entrega da versão final do software. Em vez de enfrentar os atrasos no projeto, o gerente pode considerar que a entrega de um sistema incompleto ou de baixa qualidade é melhor do que nada. Contudo, isso geralmente não é sensato pelas seguintes razões:

1. Pode ser impossível ajustar o protótipo para atender aos requisitos não funcionais ignorados durante o desenvolvimento do protótipo, tais como desempenho, segurança, robustez e confiabilidade.
2. Mudanças rápidas durante o desenvolvimento significam inevitavelmente que o protótipo não está documentado. A única especificação do projeto é o código do protótipo. Isso não é suficiente para manutenções em longo prazo.
3. As mudanças feitas durante o desenvolvimento do protótipo terão provavelmente degradado a estrutura do sistema. Será difícil e caro mantê-lo.
4. Padrões de qualidade organizacionais são geralmente relegados a segundo plano no desenvolvimento do protótipo.

Figura 17.11

Processo de desenvolvimento de protótipos.

Os protótipos *throw-away* não têm que ser executáveis para serem úteis no processo de engenharia de requisitos. Como explicado no Capítulo 16, esboços em papel da interface com o usuário do sistema (Rettig, 1994) podem ser eficientes ao ajudarem os usuários a refinar um projeto de interface e trabalhar em cenários de utilização. O desenvolvimento desses esboços é muito barato e podem ser criados em poucos dias. Uma extensão dessa técnica é um protótipo Mágico de Oz no qual somente a interface com o usuário é desenvolvida. Os usuários interagem com essa interface, mas suas solicitações são passadas para uma pessoa que as interpreta e produz a resposta apropriada.

PONTOS-CHAVE

- Com o crescimento da pressão para a entrega rápida de software, a abordagem iterativa para o desenvolvimento do software está se tornando cada vez mais usada como técnica-padrão de desenvolvimento de sistemas pequenos e médios, especialmente no mundo dos negócios.
- Os métodos ágeis são métodos iterativos de desenvolvimento que enfocam a especificação, projeto e implementação de sistemas incrementais. Eles envolvem diretamente o cliente no processo de desenvolvimento. A redução de overhead de desenvolvimento pode fazer o desenvolvimento de software ser o mais rápido possível.
- A *extreme programming* é um método ágil bastante conhecido que integra uma gama de boas práticas de programação, tais como teste sistemático, aprimoramento contínuo do software e participação do cliente na equipe de desenvolvimento.
- Um ponto forte em particular da *extreme programming* é o desenvolvimento de testes automatizados antes da criação de um recurso do programa. Todos os testes deverão ser executados com sucesso quando um incremento for integrado ao sistema.
- O desenvolvimento rápido de aplicações envolve o uso de ambientes de desenvolvimento que incluem ferramentas poderosas para apoiar a produção de software. Estas incluem linguagens de programação de banco de dados, geradores de formulários e relatórios, e links para as aplicações de escritório.
- A prototipação *throw-away* é um processo de desenvolvimento iterativo no qual um protótipo de sistema é usado para explorar as opções de requisitos e de projeto. Esse protótipo não se destina à implantação pelo cliente.
- Ao implementar um protótipo *throw-away*, são desenvolvidas primeiro as partes do sistema menos compreendidas; por outro lado, em uma abordagem incremental, começa-se pelo desenvolvimento das partes do sistema mais bem compreendidas.

LEITURAS SUGERIDAS

Extreme programming explained. Este foi o primeiro livro sobre XP e é ainda, talvez, o mais lido. Ele explica a abordagem da perspectiva de um dos criadores, e o entusiasmo é notório. (Kent Beck, 2000, Addison-Wesley.)

"Get ready for agile methods, with care". Uma crítica inteligente dos métodos ágeis que explica os pontos fortes e fracos, escrita por um engenheiro de software com vasta experiência (B. Boehm, *IEEE Computer*, jan. 2002.)

"Scripting: higher-level programming for the 21st century". Uma visão geral sobre as linguagens de scripting pelo criador do Tcl/Tk, que explica as vantagens dessa abordagem para o desenvolvimento rápido de aplicações (J. K. Ousterhout, *IEEE Computer*, mar. 1998.)

DSDM: dynamic systems development method. Descrição de abordagem para desenvolvimento rápido de aplicações que algumas pessoas consideram ser uma das primeiras instâncias de um método ágil. (J. Stapleton, 1997, Addison-Wesley.)

EXERCÍCIOS

17.1 Explique por que a entrega e a implantação rápidas de novos sistemas são freqüentemente mais importantes para os negócios do que a funcionalidade detalhada desses sistemas.

17.2 Explique como os princípios em que se baseiam os métodos ágeis conduzem ao desenvolvimento e implantação acelerados do software.

17.3 Quando você seria contra o uso de um método ágil para o desenvolvimento de um sistema de software?

17.4 A *extreme programming* expressa os requisitos de usuário como história, sendo cada história escrita em um cartão. Explique as vantagens e as desvantagens dessa abordagem para a descrição de requisitos.

17.5 Explique por que o desenvolvimento *test-first* auxilia o programador a compreender melhor os requisitos de sistema. Quais são as dificuldades potenciais do desenvolvimento *test-first*?

17.6 Sugira quatro motivos pelos quais a taxa de produtividade dos programadores que trabalham em pares é aproximadamente a mesma que quando dois programadores trabalham individualmente.

17.7 Foi solicitado que você investigue a viabilidade de prototipação no processo de desenvolvimento de software na sua organização. Escreva um relatório para seu gerente que explique os tipos de projetos nos quais a prototipação deve ser usada e estabeleça os custos e benefícios esperados da prototipação.

17.8 Um gerente de software está desenvolvendo um sistema de apoio ao projeto de software que auxilia na conversão dos requisitos de software em uma especificação formal. Comente sobre as vantagens e as desvantagens das seguintes estratégias de desenvolvimento:

a. Desenvolver um protótipo *throw-away*, avaliá-lo e, então, revisar os requisitos de sistema. Desenvolver o sistema final usando C.

b. Desenvolver o sistema a partir de requisitos existentes usando Java, então modificá-lo para adaptá-lo a quaisquer requisitos alterados pelo usuário.

c. Desenvolver o sistema usando o desenvolvimento incremental com um usuário na equipe de desenvolvimento.

17.9 Uma instituição de caridade solicitou que você crie um protótipo de um sistema que controle as doações recebidas. Esse sistema deve manter os nomes e os endereços dos doadores, seus interesses particulares, a quantidade doada e quando a doação foi feita. Se os donativos estiverem acima de um determinado valor, o doador poderá estabelecer condições para a doação (por exemplo, que ela deve ser empregada em um determinado projeto), e o sistema deve controlar como a doação foi empregada. Explique como você criaria o protótipo desse sistema, tendo em mente que a instituição de caridade possui funcionários assalariados e voluntários. Muitos dos voluntários são aposentados com pouca ou nenhuma experiência com computadores.

17.10 Você desenvolveu um sistema de protótipo *throw-away* para um cliente que está muito satisfeito com ele. Contudo, esse cliente sugere que não há necessidade de desenvolver outro sistema, mas você deve entregar o protótipo. O cliente oferece a você um excelente preço pelo sistema. Você sabe que pode haver problemas futuros com a manutenção do sistema. Explique como você pode responder a esse cliente.

18

Reuso de software

Objetivos

Os objetivos deste capítulo são apresentar o reuso de software e explicar como ele contribui para o processo de desenvolvimento de software. Depois de ler este capítulo você:

- compreenderá os benefícios e os problemas do reuso de software ao desenvolver novos sistemas;
- aprenderá várias maneiras de implementar o reuso de software;
- compreenderá o *concept reuse* e como esses conceitos reusáveis podem ser representados como padrões ou incorporados em geradores de programas;
- aprenderá como os sistemas podem ser desenvolvidos rapidamente pela composição de grandes aplicações prontas;
- conhecerá as linhas de produto de software compostas de uma arquitetura de núcleo comum e componentes configuráveis e reusáveis.

Conteúdo

18.1 O panorama do reuso
18.2 Padrões de projeto
18.3 Reuso baseado em geradores
18.4 Frameworks de aplicações
18.5 Reuso do sistema de aplicações

O processo de projeto, na maioria das disciplinas de engenharia, baseia-se no reuso de sistemas existentes ou componentes. Engenheiros mecânicos ou elétricos normalmente não especificam um projeto em que cada componente tenha de ser fabricado especialmente. Eles baseiam seus projetos em componentes já experimentados e testados em outros sistemas. Esses componentes não são apenas pequenos componentes como flanges e válvulas, mas incluem subsistemas maiores, como motores, condensadores ou turbinas.

A engenharia de software baseada em reuso é uma estratégia de comparação de engenharia de software na qual o processo de desenvolvimento é voltado ao reuso do software existente. Embora os benefícios do reuso tenham sido reconhecidos por muitos anos (McIlroy, 1968), somente nos últimos dez anos houve uma transição gradual do desenvolvimento de software original para o desenvolvimento baseado em reuso. O movimento para o desenvolvimento baseado em reuso foi uma resposta às demandas por menores custos de produção e manutenção de software, entregas

mais rápidas de sistemas e aumento da qualidade do software. Cada vez mais empresas vêem seu software como um ativo valioso e estão promovendo o reuso para aumentar seu retorno sobre investimentos em software.

A engenharia de software baseada em reuso é uma abordagem de desenvolvimento que tenta maximizar o reuso do software existente. As unidades de software reusadas podem ser de tamanhos radicalmente diferentes. Por exemplo:

1. *Reuso do sistema da aplicação.* Todo o sistema da aplicação pode ser reusado por incorporação a outros sistemas sem mudanças, pela configuração da aplicação para diferentes clientes ou pelo desenvolvimento de famílias de aplicações com uma arquitetura comum, mas adaptadas a clientes específicos. O reuso do sistema da aplicação será abordado na Seção 18.5.
2. *Reuso de componentes.* Componentes de uma aplicação que variam desde subsistemas a simples objetos podem ser reusados. Por exemplo, um sistema de identificação de padrões desenvolvido como parte de um sistema de processamento de texto pode ser reusado em um sistema de gerenciamento de banco de dados. Esse assunto será tratado no Capítulo 19.
3. *Reuso de objetos e funções.* Componentes de software que implementam uma única função, tal como uma função matemática ou uma classe de objeto, podem ser reusados. Essa forma de reuso, baseada em bibliotecas padronizadas, foi comum nos últimos 40 anos. Muitas bibliotecas de funções e classes para tipos diferentes de aplicações e plataformas de desenvolvimento estão disponíveis. Elas podem ser facilmente usadas por vinculação ao código de outra aplicação. Em áreas tal como algoritmos e gráficos matemáticos, nas quais é necessário conhecimento específico para desenvolver objetos e funções, essa é uma abordagem particularmente eficiente.

Sistemas e componentes de software são entidades específicas reusáveis, mas sua natureza específica significa algumas vezes que é caro modificá-los para uma nova situação. Uma forma complementar de reuso é o *concept reuse,* na qual, mais que um componente, a entidade reusada é mais abstrata e é projetada para ser configurada e adaptada a uma gama de situações. O concept reuse pode ser incorporado a abordagens como design patterns, produtos de sistema configuráveis e geradores de programas. O processo de reuso, quando relacionado a conceitos, inclui uma atividade de instanciação na qual conceitos abstratos são configurados para uma situação específica. Eu explico duas dessas abordagens de concept reuse — design patterns e geração de programas — mais adiante neste capítulo.

Uma vantagem óbvia do reuso de software é que os custos totais de desenvolvimento são reduzidos. Poucos componentes de software precisam ser especificados, projetados, implementados e validados. Contudo a redução de custos é somente uma das vantagens do reuso. Na Tabela 18.1, listei outras vantagens do reuso de ativos de software.

Há também custos e problemas associados ao reuso (Tabela 18.2). Em particular há um custo significativo associado ao entendimento de se um componente é adequado para reuso em uma determinada situação e ao teste de se o componente é confiável. Esses custos adicionais podem inibir a introdução do reuso e significar que a redução de custos totais de desenvolvimento pode ser menor que a prevista.

Tabela 18.1 Benefícios do reuso de software

Benefício	Explicação
Confiança aumentada	Software reusado, experimentado e testado em sistemas de trabalho, deve ser mais confiável do que software novo, pois seus defeitos de projeto e implementação já foram encontrados e corrigidos.
Risco de processo reduzido	O custo de software existente já é conhecido, enquanto os custos de desenvolvimento são sempre um problema de avaliação. Isso é um fator importante para o gerenciamento de projetos porque reduz a margem de erro de estimativa de custo de projeto. Tal fato é especialmente verdadeiro quando componentes de software relativamente grandes, tais como subsistemas, são reusados.
Uso eficiente de especialistas	Em vez de fazer o mesmo trabalho repetidas vezes, esses especialistas podem desenvolver software reusável englobando seus conhecimentos.
Conformidade com padrões	Alguns padrões, tais como padrões de interface com o usuário, podem ser implementados como um conjunto de componentes reusáveis padronizados. Por exemplo, se os menus numa interface com o usuário são implementados usando componentes reusáveis, todas as aplicações apresentarão o mesmo formato de menu para o usuário. O uso de interfaces padronizadas com o usuário melhora a confiança porque os usuários provavelmente cometerão menos erros quando apresentados a uma interface familiar.
Desenvolvimento acelerado	A apresentação de um sistema para o mercado tão cedo quanto possível é muitas vezes mais importante do que os custos totais de desenvolvimento. O reuso de software pode tornar rápida a produção do sistema porque tanto o tempo de desenvolvimento quanto o de validação devem ser reduzidos.

Tabela 18.2 Problemas com reuso

Benefício	Explicação
Custos de manutenção aumentados	Se o código-fonte de um sistema ou componente de software reusável não estiver disponível, então os custos de manutenção poderão ser aumentados, pois os elementos reusados do sistema podem tornar-se cada vez mais incompatíveis com as mudanças.
Falta de apoio de ferramenta	Os conjuntos de ferramentas CASE podem não apoiar o desenvolvimento com reuso. Pode ser difícil ou impossível integrar essas ferramentas a um sistema de biblioteca de componentes. O processo de software suposto por essas ferramentas pode não levar em conta o reuso.
Síndrome do não-inventado-aqui	Alguns engenheiros de software preferem reescrever componentes porque acreditam que podem aprimorá-los. Isso tem, em parte, a ver com responsabilidade e, em outra, com o fato de que a escrita do software original é vista como mais desafiadora do que reusar software de outras pessoas.
Criação e manutenção de uma biblioteca de componentes	Preencher uma biblioteca de componentes reusáveis e assegurar aos desenvolvedores de software que podem usar essa biblioteca pode ser caro. Nossas técnicas atuais de classificação, catalogação e recuperação de componentes de software são imaturas.
Procura, compreensão e adaptação de componentes reusáveis	Componentes de software precisam ser descobertos numa biblioteca, compreendidos e, às vezes, adaptados para trabalhar num novo ambiente. Os engenheiros devem estar razoavelmente confiantes de encontrar um componente na biblioteca antes de incluírem uma busca de componente como parte de seu processo normal de desenvolvimento.

O reuso sistemático não acontece simplesmente — ele deve ser planejado e introduzido por meio de um programa que envolva toda a organização. Isso foi reconhecido há muitos anos no Japão (Matsumoto, 1984), onde o reuso é uma parte integrante da abordagem de 'fábrica' japonesa para o desenvolvimento de software (Cusamano, 1989). Empresas como Hewlett-Packard também têm sido muito bem-sucedidas em seus programas de reuso (Griss e Wosser, 1995), e sua experiência foi incorporada em um livro comum por Jacobsen et al. (Jacobsen, et al., 1997).

18.1 O panorama do reuso

Nos últimos 20 anos, muitas técnicas foram desenvolvidas para apoiar o reuso de software. Elas exploraram o fato de que sistemas em um mesmo domínio de aplicação são similares e têm potencial de reuso, que o reuso é possível em diferentes níveis (a partir de funções simples até aplicações completas) e que padrões para componentes reusáveis facilitam o reuso. A Figura 18.1 mostra o número de maneiras de apoiar o reuso de software; cada uma é brevemente descrita na Tabela 18.3.

Dado esse conjunto de técnicas de reuso, a questão principal é: qual a técnica mais apropriada? Obviamente, isso depende dos requisitos do sistema que será desenvolvido, da tecnologia e dos ativos reusáveis disponíveis e do conhecimento da equipe de desenvolvimento. Fatores principais a serem considerados no planejamento do reuso são:

Figura 18.1

Panorama de reuso.

Tabela 18.3 Abordagens que apóiam o reuso de software

Abordagem	Descrição
Design patterns	Abstrações genéricas que ocorrem ao longo das aplicações são representadas como design patterns que mostram objetos abstratos e concretos e interações.
Desenvolvimento baseado em componentes	Sistemas desenvolvidos pela integração de componentes (conjuntos de objetos) que estão em conformidade com padrões de modelos e componentes. Isso é explicado no Capítulo 19.
Frameworks de aplicação	Conjuntos de classes abstratas e concretas podem ser adaptados e ampliados para criar sistemas de aplicações.
Empacotamento de sistemas legados	Sistemas legados (veja Capítulo 2) que podem ser 'empacotados' pela definição de um conjunto de interfaces e fornecimento de acesso a esses sistemas herdados através das interfaces.
Sistemas orientados a serviços	Sistemas desenvolvidos pela ligação de serviços compartilhados, que podem ser providos externamente.
Linhas de produtos de aplicação	Um tipo de aplicação generalizada com base em uma arquitetura comum de tal maneira que possa ser adaptada para clientes diferentes.
Integração de COTS	Sistemas desenvolvidos pela integração de sistemas de aplicações existentes.
Aplicações verticais configuráveis	Um sistema genérico projetado de tal maneira que pode ser configurado para as necessidades de clientes de sistemas específicos.
Bibliotecas de programa	Bibliotecas de classes e funções que implementam abstrações comumente usadas disponíveis para reuso.
Geradores de programa	Um sistema gerador que incorpora conhecimento de um determinado tipo de aplicação e pode gerar sistemas ou fragmentos de sistema no domínio.
Desenvolvimento de software orientado a aspectos	Componentes compartilhados integrados em uma aplicação em diferentes lugares quando o programa é compilado.

1. *Cronograma de desenvolvimento do software.* Se o software tem de ser desenvolvido rapidamente, você deve tentar reusar sistemas prontos em vez de componentes individuais. Esses são grandes ativos reusáveis. Embora a adequação aos requisitos possa ser imperfeita, essa abordagem minimiza a quantidade de desenvolvimento necessária.
2. *Ciclo de vida previsto do software.* Se você está desenvolvendo um sistema de ciclo de vida longo, você deve se concentrar na facilidade de manutenção do sistema. Nessas circunstâncias, você não deve pensar apenas nas possibilidades imediatas de reuso, mas também nas implicações no longo prazo. Você terá de adaptar o sistema aos novos requisitos, que provavelmente significarão mudanças nos componentes e em como eles são usados. Se você não tem acesso ao código-fonte, evite o uso de componentes e sistemas de fornecedores externos; não é possível ter certeza de que esses fornecedores serão capazes de continuar a fornecer suporte ao software reusado.
3. *Conhecimento, habilidades e experiência da equipe de desenvolvimento.* Todas as tecnologias de reuso são bastante complexas e você precisa de bastante tempo para compreender e usá-las eficientemente. Portanto, se a equipe de desenvolvimento tem habilidades em uma determinada área, esse é provavelmente o ponto a ser enfocado.
4. *Importância do software e seus requisitos não funcionais.* Para um sistema crítico que precisa ser certificado por regulamentação externa, pode ser necessário criar um caso de confiança para o sistema (explicado no Capítulo 24). Isso será difícil se você não tiver acesso ao código-fonte do software. Se seu software tem requisitos de desempenho severos, pode ser impossível usar estratégias tal como reuso nos geradores de programas. Esses sistemas tendem a gerar código relativamente ineficiente.
5. *Domínio da aplicação.* Em alguns domínios de aplicação, tal como sistemas de informações industriais e médicos, há vários produtos genéricos que podem ser reusados por meio de configuração para uma situação específica. Se você estiver trabalhando em tal domínio, considere sempre esses produtos como uma opção.
6. *Plataforma sobre a qual o sistema será executado.* Alguns modelos de componentes, tal como COM/Active X, são específicos para plataformas Microsoft. Se o desenvolvimento ocorrer sobre tal plataforma, essa pode ser a abordagem mais adequada. Da mesma forma, sistemas de aplicação genéricos podem ser específicos em plataforma e você pode somente ser capaz de reusá-los se seu sistema for projetado para a mesma plataforma.

A gama de técnicas de reuso disponíveis é tal que, na maioria das situações, há a possibilidade de algum reuso de software. Empregar o reuso é freqüentemente uma decisão mais gerencial do que técnica. Os gerentes podem não desejar comprometer seus requisitos ao permitir o uso de componentes reusáveis ou eles podem decidir que o desenvolvimento de componentes originais ajudará a criar uma base de ativos de software. Eles podem não compreender os riscos associados ao reuso da mesma forma que compreendem os riscos de desenvolvimento original. Portanto, embora os riscos de desenvolvimento de novo software possam ser maiores, alguns gerentes podem preferir riscos conhecidos a desconhecidos.

18.2 Design patterns

Ao tentar reusar componentes executáveis, você estará inevitavelmente limitado pelas decisões de projeto detalhado feitas pelos implementadores desses componentes. Elas variam de algoritmos específicos usados para implementar os componentes aos objetos e tipos nas interfaces do componente. Quando essas decisões de projeto entram em conflito com seus requisitos específicos, o reuso de componentes é impossível ou introduz ineficiências no seu sistema.

Uma maneira de contornar isso é reusar projetos abstratos que não incluem detalhes de implementação. Você pode implementar esses projetos para atender aos requisitos de sua aplicação específica. As primeiras instâncias dessa abordagem de reuso vieram com a documentação e publicação de algoritmos fundamentais (Knuth, 1971) e, mais tarde, com a documentação de tipos de dados abstratos tais como pilhas, árvores e listas (Booch, 1987). Mais recentemente, essa abordagem de reuso foi incorporada aos padrões de projeto.

Os design patterns foram derivados das idéias apresentadas por Christopher Alexander (Alexander, et al., 1977), que sugeriu haver certos padrões no projeto de criação que eram comuns e inerentemente agradáveis e eficientes. O padrão é uma descrição do problema e a essência da sua solução, dessa forma a solução pode ser reusada em aplicações diferentes. O padrão não é uma especificação detalhada. Em vez disso, você pode pensar nele como uma descrição de conhecimento e experiência acumulados, uma solução comprovada para um problema comum. Uma citação do site Web *hillside.net*, dedicado a manter informações sobre padrões, engloba seu papel no reuso:

> *Padrões e linguagens de padrões são maneiras de descrever melhores práticas, bons projetos, e captam a experiência de uma maneira possível de ser reusada por outros.*

A maioria dos projetistas pensa em padrões de projeto como uma maneira de apoiar o projeto orientado a objetos. Os padrões freqüentemente contam com características de objetos tais como herança e polimorfismo para fornecer a generalidade. Contudo, o princípio geral de englobamento de experiência em um padrão é igualmente aplicável a todas as abordagens de projeto de software.

Gamma et al. (Gamma, et al., 1995) definem os quatro elementos essenciais de design patterns:

1. Um nome que seja uma referência significativa para o padrão.
2. Uma descrição da área do problema que explique quando o padrão pode ser aplicado.
3. Uma descrição da solução das partes da solução de projeto, seus relacionamentos e suas responsabilidades. Isso não é uma descrição de projeto concreta. É um template que pode ser instanciado de diferentes maneiras. Ele é muitas vezes expresso graficamente e mostra os relacionamentos entre os objetos e as classes de objetos na solução.
4. Uma declaração das conseqüências — os resultados e compromissos — da aplicação do padrão. Isso pode auxiliar os projetistas a compreender se um padrão pode ser efetivamente aplicado em uma determinada situação.

Esses elementos essenciais de uma descrição de padrão podem ser decompostos, conforme mostrado no Quadro 18.1. Por exemplo, Gamma e seus co-autores dividiram a descrição do problema em motivação (uma descrição do motivo de o padrão ser útil) e aplicabilidade (uma descrição de situações nas quais o padrão pode ser usado). Abaixo da descrição da solução, eles descrevem a estrutura do padrão, participantes, colaborações e implementação.

Para ilustrar a descrição do padrão, uso o padrão Observer, extraído do livro de Gamma et al. Esse padrão pode ser usado em várias situações em que são necessárias apresentações diferentes de um estado de objeto. Ele separa o objeto que deve ser exibido das diferentes formas de apresentação. Isso é ilustrado na Figura 18.2, que mostra duas apresentações gráficas do mesmo conjunto de dados. Na minha descrição, uso os quatro elementos essenciais de descrição e complemento-os com uma breve instrução do que o padrão pode fazer.

As representações gráficas são normalmente usadas para ilustrar as classes de objetos usadas em padrões e seus relacionamentos. Elas complementam a descrição do padrão e adicionam detalhes à descrição da solução. A Figura 18.3 é a representação em UML do padrão Observer.

Um número grande de padrões publicados está disponível atualmente (consulte o site Web do livro para obter os links), abrangendo uma gama de domínios de aplicações e de linguagens. A noção de um padrão como um conceito

Quadro 18.1

Descrição do padrão Observer.

Nome do padrão: Observer

Descrição: Separa o display de estado de um objeto a partir do objeto em si e permite que displays alternativos sejam fornecidos. Quando o estado do objeto muda, todos os diplays são automaticamente notificados e atualizados para refletir a mudança.

Descrição do problema: Em muitas situações, é necessário fornecer vários displays de alguma informação de estado, tais como um display gráfico e um display tabular. Nem todos eles podem ser conhecidos quando a informação é especificada. Todas as apresentações alternativas podem supor interação e, quando o estado é alterado, todos os displays devem ser atualizados.

Esse padrão pode ser usado em todas as situações em que mais de um formato de display para a informação de estado pode ser necessário e em que não é necessário para o objeto que mantém a informação de estado saber sobre os formatos específicos de displays usados.

Descrição da solução: A estrutura do padrão é mostrada na Figura 18.7. Esta define dois objetos abstratos, Subject e Observer, e dois objetos concretos, ConcreteSubject e ConcreteObject, que herdam os atributos dos objetos abstratos relacionados. O estado a ser mostrado é mantido em ConcreteObject, que também herda operações de Subject permitindo-lhe adicionar e remover Observers e emitir uma notificação quando o estado foi alterado.

ConcreteObserver mantém uma cópia do estado do ConcreteSubject e implementa a interface Update () do Observer que permite que essas cópias sejam mantidas na etapa. ConcreteObserver mostra automaticamente seu estado — isso não é normalmente uma operação de interface.

Conseqüências: Subject conhece apenas o Observer abstrato e não conhece detalhes da classe concreta. Portanto, há um acoplamento mínimo entre esses objetos. Por causa dessa falta de conhecimento, as otimizações que aprimoram o desempenho do display são impraticáveis. Mudanças em Subject podem fazer com que um conjunto de atualizações ligadas a Observers sejam geradas, sendo que alguma delas podem não ser necessárias.

Figura 18.2

Vários displays.

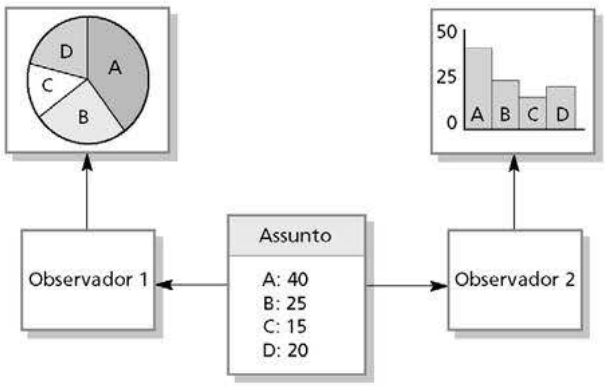
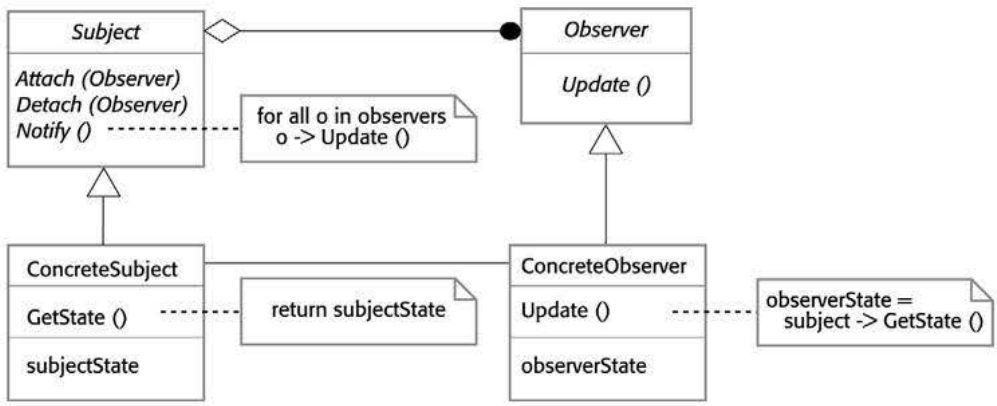


Figura 18.3

Padrão Observer.



reusável foi desenvolvida em várias áreas fora do projeto de software, incluindo o gerenciamento de configuração, projeto de interface com o usuário e cenários de interação (Berczuk e Appleton, 2002; Borchers, 2001; Martin, et al., 2001; Martin, et al., 2002).

O uso de padrões é uma forma eficiente de reuso. Contudo, estou convencido de que somente engenheiros de software experientes com profundo conhecimento de padrões conseguem usá-los eficientemente. Esses desenvolvedores podem reconhecer

situações genéricas nas quais um padrão possa ser aplicado. Programadores inexperientes, mesmo que tenham lido os livros sobre padrões, terão dificuldade em decidir se podem reusar um padrão ou se é necessário desenvolver uma solução específica.

18.3 Reuso baseado em geradores

O concept reuse por meio de padrões conta com a descrição do conceito de uma maneira abstrata e sua disponibilização para o desenvolvedor de software criar uma implementação. Uma abordagem alternativa é o reuso baseado em geradores (Biggerstaff, 1988). Nessa abordagem, o conhecimento reusável é capturado em um sistema gerador de programas que pode ser programado por especialistas em domínio usando uma linguagem orientada a domínios ou uma ferramenta CASE interativa que suporte geração de sistemas. A descrição da aplicação especifica, de maneira abstrata, quais componentes reusáveis serão empregados, como eles serão combinados e sua parametrização. Usando essas informações, um sistema de software operacional pode ser gerado (Figura 18.4).

O reuso baseado em geradores beneficia-se do fato de que aplicações no mesmo domínio, tal como sistemas de negócios, têm arquiteturas comuns e realizam funções comparáveis. Por exemplo, conforme expliquei no Capítulo 13, os sistemas de processamento de dados normalmente seguem um modelo entrada-processo-saída e, geralmente, incluem operações como verificação de dados e geração de relatórios. Portanto, componentes genéricos para seleção de itens de um banco de dados, verificação de se eles estão dentro de um intervalo e a criação de relatórios podem ser criados e incorporados em um gerador de aplicações. Para reusar esses componentes, o programador simplesmente deve selecionar os itens de dados a serem usados, as verificações a serem aplicadas e a formatação dos relatórios.

O reuso baseado em geradores foi particularmente bem-sucedido em sistemas de aplicações de negócios e há muitos produtos geradores de aplicações de negócios diferentes disponíveis. Eles podem gerar aplicações completas ou automatizar parcialmente a criação de aplicações, deixando que o programador preencha os detalhes específicos. A abordagem baseada em geradores para reuso também é usada em outras áreas, incluindo:

1. *Geradores de parser para processamento de linguagem.* A entrada do gerador é uma gramática que descreve a linguagem a ser analisada e a saída é um parser da linguagem. Essa abordagem está incorporada a sistemas como *lex* e *yacc* para C e JavaCC, um compilador de Java.
2. *Geradores de código em ferramentas CASE.* A entrada desses geradores é um projeto de software e a saída é um programa que implementa o sistema projetado. Esses geradores podem ser baseados em modelos UML e, dependendo das informações contidas nos modelos UML, geram um programa ou componente completo, ou um esqueleto de código. O desenvolvedor de software, então, adiciona detalhes para completar o código.

Essas abordagens para reuso baseado em geradores beneficiam-se da estrutura comum das aplicações nessas áreas. A técnica também foi usada em domínios de aplicações mais específicos, tais como sistemas de comando e controle (O'Connor, et al., 1994) e instrumentação científica (Butler, 1994), nos quais bibliotecas de componentes foram desenvolvidas. Especialistas em domínio, então, usam uma linguagem específica de domínio para desenvolver esses componentes e criam aplicações. Contudo, há um custo inicial alto na definição e implementação de conceitos de domínio e na linguagem de composição. Isso significa que muitas empresas relutam em correr os riscos de adotar essa abordagem.

O reuso baseado em geradores tem custo adequado para aplicações tais como processamento de dados de negócios. É muito mais fácil para os usuários finais desenvolver programas que usam geradores em comparação a outras abordagens baseadas em componentes para reuso. Inevitavelmente, contudo, há ineficiências nos programas gerados. Isso significa que pode ser impossível usar essa abordagem em sistemas com requisitos de alto desempenho e throughput.

A programação baseada em geradores é um componente importante das técnicas emergentes de desenvolvimento de software que combina geração de programas com o desenvolvimento baseado em componentes. O livro de Czarnecki e Eisenecher (Czarnecki e Eisenecher, 2000) descreve essas novas abordagens.

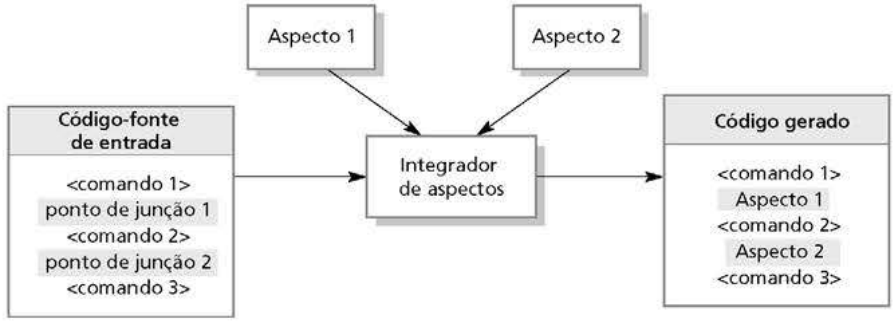
A mais desenvolvida dessas abordagens é o desenvolvimento de software orientado a aspectos (AOSD — Aspect-Oriented Software Development) (Elrad, et al., 2001). Essa abordagem soluciona um dos principais problemas do projeto de software — o problema da separação de assuntos. A separação de assuntos é um princípio básico de projeto; você deve

Figura 18.4

Reuso baseado em geradores.

Figura 18.5

Integração de aspectos.



projetar seu software de tal maneira que cada unidade ou componente realize uma, e somente uma, função. Por exemplo, no sistema LIBSYS, deve haver um componente dedicado à busca de documentos, um componente dedicado à impressão de documentos, um componente dedicado ao gerenciamento de downloads e assim por diante.

Contudo, em muitas situações, os assuntos não estão associados a funções de aplicação claramente definidas, mas são transversais — ou seja, eles afetam todos os componentes do sistema. Por exemplo, digamos que você queira manter o acompanhamento do uso de cada um dos módulos de sistema de cada usuário. Portanto, você tem um assunto de monitoração que precisa estar associado a todos os componentes. Isso não pode ser simplesmente implementado como um objeto referenciado por esses componentes. A monitoração específica realizada necessita de informações de contexto da função de sistema que está sendo monitorada.

Na programação orientada a aspectos, esses assuntos transversais são implementados como aspectos e, dentro do programa, você define onde um aspecto deve ser associado. Eles são chamados de *pontos de junção*. Os aspectos são desenvolvidos separadamente e, então, numa etapa de pré-compilação chamada *integração de aspectos*, eles são ligados aos pontos de junção (Figura 18.5). A integração de aspectos é uma forma de geração de programas — a saída do integrador é um programa em que o código do aspecto foi integrado. Um desenvolvimento de Java chamado AspectJ (Kiczales, et al., 2001) é a linguagem mais conhecida para o desenvolvimento orientado a aspectos.

O AOSD é um tópico de pesquisa importante, mas ainda não foi largamente usado no desenvolvimento de software industrial. Há problemas com essa abordagem — verificação e validação são difíceis e precisamos de uma compreensão melhor do relacionamento entre aspectos e propriedades de sistemas não funcionais. Contudo, o AOSD está se desenvolvendo rapidamente como uma nova e importante técnica de engenharia de software. Portanto, incluí um capítulo sobre esse tópico na nova seção sobre Tecnologias Emergentes.

18.4 Frameworks de aplicações

Os proponentes iniciais do desenvolvimento orientado a objetos sugeriram que objetos eram a abstração mais apropriada para reuso. Contudo, experiências mostraram que objetos são muitas vezes pequenos e especializados demais para uma aplicação específica. Em vez disso, tornou-se claro que o reuso orientado a objetos é mais bem apoiado por um processo de desenvolvimento orientado a objetos por meio de abstrações maiores chamadas *frameworks*.

Um framework (ou framework de aplicação) é um projeto de subsistema composto por um conjunto de classes abstratas e concretas e as interfaces entre elas (Wirfs-Brock e Johnson, 1990). Detalhes específicos do subsistema da aplicação são implementados pela adição de componentes e pelo fornecimento de implementações concretas de classes abstratas no framework. Os frameworks raramente são aplicações propriamente ditas. As aplicações são geralmente constituídas pela integração de vários frameworks.

Fayad e Schmidt (Fayad e Schmidt, 1997) explicam três classes de frameworks:

1. *Frameworks de infra-estrutura de sistemas.* Esses frameworks apóiam o desenvolvimento de infra-estruturas de sistemas, tais como comunicações, interfaces com o usuário e compiladores (Schmidt, 1997).
2. *Frameworks de integração de middleware.* Consistem em um conjunto de padrões e classes de objetos associados que apóiam a comunicação e troca de informações de componentes. Exemplos desse tipo de framework incluem CORBA, COM+ da Microsoft e Enterprise Java Beans. Esses frameworks apóiam modelos de componentes padronizados, conforme explicado no Capítulo 19.
3. *Frameworks de aplicações empresariais.* São dedicados a domínios de aplicações específicos tais como sistemas de telecomunicações ou financeiros (Baumer, et al., 1997). Eles incorporam conhecimento de domínio e apóiam o desenvolvimento de aplicações de usuários finais.

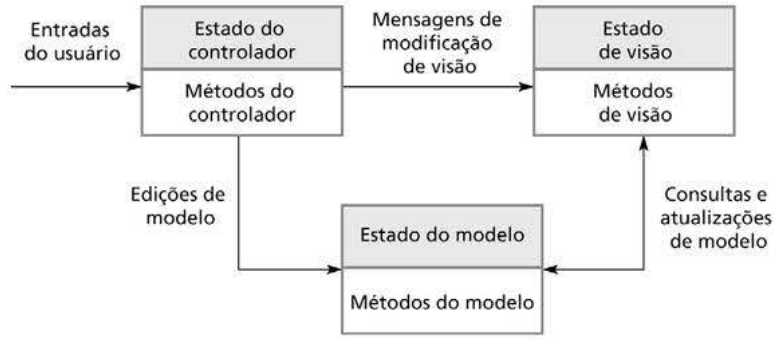


Figura 18.6

O framework Modelo-Visão-Controlador.

Conforme o nome sugere, um framework é uma estrutura genérica que pode ser ampliada para criar um subsistema ou aplicação mais específica. É implementado como um conjunto de classes de objetos abstratas ou concretas. Ampliar o framework pode envolver a adição de classes concretas que herdam operações de classes abstratas no framework. Além disso, pode ser necessário definir *callbacks*. Estes são métodos chamados em resposta a eventos reconhecidos pelo framework.

Um dos mais conhecidos e mais amplamente usados frameworks para projeto de GUI é o framework Modelo-Visão-Controlador (MVC — Model-View-Controller) (Figura 18.6). O framework MVC foi originalmente proposto na década de 80 como uma abordagem para projeto de GUI que permitia diversas apresentações de um objeto e estilos separados de interação com essas apresentações. O framework MVC apóia a apresentação de dados de maneiras diferentes (veja a Figura 18.2) e a interação separada com cada uma dessas apresentações. Quando os dados são modificados através de uma das apresentações, todas as outras são atualizadas.

Os frameworks são muitas vezes instanciações de um número de padrões, conforme explicado na Seção 18.2. Por exemplo, o framework MVC inclui o padrão Observer descrito no Quadro 18.1, o padrão Strategy dedicado à atualização do modelo, o padrão Composite e vários outros explicados por Gamma et al. (Gamma, et al., 1995).

As aplicações construídas usando frameworks podem ser a base para reuso posterior, por meio do conceito de linhas de produto de software ou famílias de aplicações, conforme explicado na Seção 18.5.2. Devido a essas aplicações serem constituídas por um framework, a modificação de membros da família para criar novos membros é simplificada. Contudo, frameworks são geralmente mais abstratos que produtos genéricos e, portanto, permitem a criação de uma ampla gama de aplicações.

O problema fundamental com os frameworks é sua complexidade inerente e o tempo necessário para aprender a usá-los. Vários meses podem ser necessários para compreender completamente um framework, portanto, é provável que, em grandes organizações, alguns engenheiros de software tornem-se especialistas em frameworks. Não há dúvidas de que essa é uma abordagem efetiva para reuso, mas é muito caro aplicá-la nos processos de desenvolvimento de software.

18.5 Reuso do sistema de aplicações

O reuso do sistema de aplicações envolve o reuso de sistemas de aplicações inteiros pela configuração de um sistema para um ambiente específico ou pela integração de dois ou mais sistemas para criar uma nova aplicação. Conforme sugeri na Seção 18.1, o reuso do sistema de aplicações é freqüentemente a técnica mais eficiente de reuso. Envolve o reuso de grandes ativos que podem ser rapidamente configurados para criar um novo sistema.

Nesta seção, explico dois tipos de reuso de aplicações: a criação de novos sistemas pela integração de duas ou mais aplicações comerciais e o desenvolvimento de linhas de produtos. Uma linha de produtos é um conjunto de sistemas baseado em uma arquitetura de núcleo comum e componentes compartilhados. O sistema central é especificamente projetado para ser configurado e adaptado para se adequar às necessidades específicas de diferentes clientes do sistema.

18.5.1 Reuso de produto COTS

Um produto comercial (COTS) é um sistema de software que pode ser usado sem alterações pelo comprador. Virtualmente, todo software de computador de mesa e uma grande variedade de produtos para servidores são softwares do tipo COTS. Devido a esse software ser projetado para uso geral, geralmente inclui muitas características e funções com potencial de reuso em aplicações e ambientes diferentes. Embora possa haver problemas com essa abordagem na construção de sistema (Tracz, 2001), há um número crescente de casos de sucesso que demonstram sua viabilidade (Baker, 2002; Balk e Kedia, 2000; Pfarr e Reis, 2002).

Alguns tipos de produtos COTS foram reusados por muitos anos. Sistemas de banco de dados são, talvez, o melhor exemplo disso. Bem poucos desenvolvedores considerariam implementar seus próprios sistemas de gerenciamento de banco de dados. Contudo, até meados da década de 1990, havia somente poucos sistemas grandes, como os sistemas de gerenciamento de banco de dados e monitores de teleprocessamento, que foram usados rotineiramente. A maioria dos sistemas foi projetada como sistemas independentes e, freqüentemente, havia muitos problemas em fazer esses sistemas trabalharem juntos.

Atualmente é comum que sistemas grandes definam Interfaces de Programação de Aplicação (APIs — Application Programming Interfaces, que permitem programar acesso às funções de sistema. Isso significa que a criação de grandes sistemas, tais como sistemas de e-commerce, por meio da integração de uma gama de sistemas COTS, deve ser sempre considerada como uma opção importante de projeto. Devido às funcionalidades que esses produtos COTS oferecem, é possível reduzir muito os custos e tempo de entrega em comparação ao desenvolvimento de um novo software. Além disso, os riscos podem ser reduzidos à medida que o produto já está disponível e os gerentes podem verificar se ele atende aos seus requisitos.

Para desenvolver sistemas usando produtos COTS, você precisa fazer algumas escolhas de projeto:

1. *Quais produtos COTS oferecem a funcionalidade mais apropriada?* Se você ainda não tem experiência com um produto COTS, pode ser difícil decidir qual produto é mais adequado.
2. *Como os dados serão trocados?* Em geral, produtos individuais usam estruturas de dados e formatos exclusivos e você precisa escrever adaptadores que convertam uma representação em outra.
3. *Quais recursos de um produto serão realmente usados?* A maioria dos produtos COTS tem mais funcionalidades do que você necessita e a funcionalidade é muitas vezes duplicada entre produtos diferentes. Você precisa decidir quais características de qual produto são mais apropriadas aos seus requisitos. Se possível, bloqueie o acesso à funcionalidade não usada porque isso pode interferir na operação normal do sistema. A falha do primeiro vôo do foguete Ariane 5, explicada no Capítulo 19 (Nuseibeh, 1997), foi conseqüência de uma falha em uma funcionalidade não usada de um subsistema reusado.

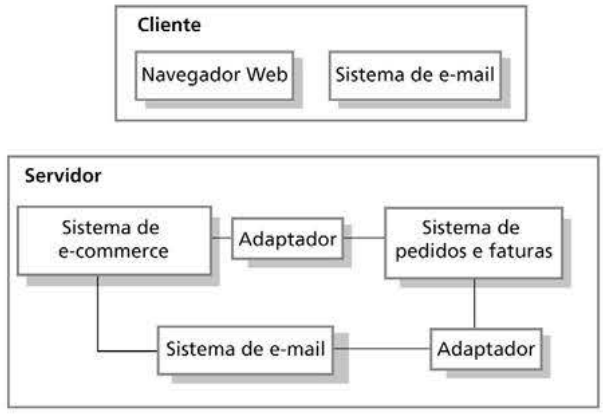
Para ilustrar a integração de COTS, digamos que uma grande organização deseja desenvolver um sistema de compras que permita às pessoas enviar pedidos de seus computadores de mesa. Com a introdução desse sistema na organização, a empresa estima a economia de cinco milhões de dólares por ano. Pela centralização de compras, o novo sistema de aquisição pode assegurar que os pedidos sejam sempre aos fornecedores que oferecem o melhor preço e deve reduzir os custos da burocracia associada aos pedidos. Como em sistemas manuais, isso envolve a escolha de produtos disponíveis de um fornecedor, criação de um pedido, aprovação do pedido, envio do pedido a um fornecedor, recebimento dos produtos e confirmação de que o pagamento deve ser feito.

A empresa tem um sistema de pedidos atual usado pela divisão de compras. Ele já está integrado ao sistema de faturas e entregas. Para criar um novo sistema de pedidos, o antigo sistema foi integrado a uma plataforma de e-commerce baseada na Web e um sistema de e-mail que lida com a comunicação com os usuários. A estrutura final do sistema de compras usando COTS é mostrada na Figura 18.7.

Esse sistema de compras baseia-se em uma arquitetura cliente–servidor, e, no cliente, são usados um navegador Web e software de e-mail padrão. Estes já estão integrados ao sistema de faturas e entregas. No servidor, a plataforma de e-commerce tem de se integrar ao sistema de pedidos atual por meio de um adaptador. O sistema de e-commerce tem seu próprio formato de pedidos, regras de entrega etc., que precisam ser convertidos no formato usado pelo sistema de pedidos. O sistema de e-commerce tem uma integração incorporada ao sistema de e-mail para enviar notificações aos usuários, mas

Figura 18.7

Sistema de aquisição baseado em COTS.



o sistema de pedidos não foi projetado para isso. Portanto, um outro adaptador precisa ser escrito para converter as notificações em mensagens de e-mail.

Em princípio, o uso de um grande sistema COTS é igual ao de qualquer outro componente mais específico. Você precisa compreender as interfaces de sistema e usá-las exclusivamente para se comunicarem com o componente; é necessário decidir entre os requisitos específicos e o desenvolvimento rápido e reuso; e você precisa projetar uma arquitetura de sistema que permita aos sistemas COTS operarem juntos.

Contudo, o fato de que esses produtos geralmente são, eles próprios, sistemas grandes e muitas vezes vendidos separadamente como sistemas independentes introduz problemas adicionais. Boehm e Abts (Boehm e Abts, 1999) expõem quatro problemas com integração de sistemas COTS:

1. *Falta de controle sobre a funcionalidade e o desempenho.* Embora a interface publicada de um produto possa parecer oferecer os recursos necessários, estes podem não ser adequadamente implementados ou podem ser executados insatisfatoriamente. O produto pode ter operações ocultas que interferem em seu uso em uma determinada situação. A correção desses problemas pode ser uma prioridade para o integrador de produtos COTS, mas pode não ser uma preocupação real para o fornecedor do produto. Os usuários podem ter de simplesmente encontrar alternativas para os problemas se desejarem reusar o produto COTS.
2. *Problemas com a interoperabilidade de sistemas COTS.* Às vezes é difícil fazer com que produtos COTS trabalhem juntos porque cada produto incorpora suas próprias suposições sobre como será usado. Garlan et al. (Garlan, et al., 1995), ao relatarem sua experiência de tentativa de integração de quatro produtos COTS, descobriram que três desses produtos baseavam-se em eventos, mas cada um usava um modelo diferente de eventos e considerava que tinha acesso exclusivo à fila de eventos. Como conseqüência, o projeto exigiu cinco vezes mais esforço que o originalmente previsto e o cronograma atrasou dois anos além dos seis meses previstos.
3. *Nenhum controle sobre a evolução do sistema.* Os fornecedores de produtos COTS tomam suas próprias decisões sobre mudanças de sistema em resposta às pressões de mercado. Em relação a produtos para PC, em particular, novas versões são freqüentemente lançadas e podem não ser compatíveis com todas as versões anteriores. Novas versões podem ter funcionalidade adicional não desejada e versões anteriores podem tornar-se indisponíveis e deixarem de contar com o suporte dos fornecedores.
4. *Suporte dos fornecedores de COTS.* O nível de suporte disponível dos fornecedores de COTS varia muito. Devido a esses sistemas serem comerciais, o suporte do fornecedor é particularmente importante quando surgem problemas, pois os desenvolvedores não têm acesso ao código-fonte e à documentação detalhada do sistema. Enquanto os fornecedores podem se comprometer a fornecer suporte, mudanças de mercado e circunstâncias econômicas podem dificultar o cumprimento desse compromisso. Por exemplo, um fornecedor de sistema COTS pode decidir descontinuar um produto devido à demanda limitada ou pode ser absorvido por outra empresa e esta não desejar oferecer suporte a todos os produtos atuais.

Naturalmente, é improvável que todos esses problemas surjam em qualquer caso, mas minha opinião é que pelo menos um deles deve ser esperado na maioria dos projetos de integração de COTS. Conseqüentemente, os benefícios de custo e cronograma do reuso de COTS serão menores do que pareciam inicialmente.

Além disto, Boehm e Abts calculam que, em muitos casos, o custo de manutenção e evolução de sistema pode ser maior quando produtos COTS são usados. Todas as dificuldades anteriores são problemas de ciclo de vida; eles não afetam somente o desenvolvimento inicial do sistema. À medida que as pessoas envolvidas na manutenção do sistema se distanciam dos desenvolvedores originais do sistema, torna-se mais provável que surjam dificuldades reais com os produtos COTS integrados.

Apesar desses problemas, eles são superados pelos benefícios do reuso de produtos COTS, pois esses sistemas oferecem muita funcionalidade para o usuário. Meses, e às vezes anos, de esforço e implementação podem ser poupados se um sistema existente for reusado e os tempos de desenvolvimento de sistemas puderem ser reduzidos drasticamente. Por exemplo, o sistema de compras descrito na Figura 18.7 foi implementado e entregue para uma grande empresa em nove meses em vez dos três anos originalmente estimados. Se a entrega rápida do sistema for essencial e houver alguma flexibilidade nos requisitos, então a integração de produtos COTS muitas vezes é a estratégia de reuso mais eficiente a ser adotada.

18.5.2 Linhas de produtos de software

Uma das abordagens mais eficazes de reuso é a criação de linhas de produtos de software ou famílias de aplicações. Uma linha de produtos é um conjunto de aplicações com uma arquitetura comum específica de aplicação, conforme explicado no Capítulo 13. Cada aplicação específica é especializada de alguma maneira. O núcleo comum da família de aplicações é reusado cada vez em que uma nova aplicação é necessária. O novo desenvolvimento pode envolver a con-

figuração de componentes específicos, a implementação de componentes adicionais e a adaptação de alguns componentes para atender às novas demandas.

Vários tipos de especialização de uma linha de produtos de software podem ser desenvolvidos:

1. *Especialização de plataforma.* Versões da aplicação são desenvolvidas para plataformas diferentes. Por exemplo, pode haver versões da aplicação para plataformas Windows, Solaris e Linux. Nesse caso, a funcionalidade da aplicação normalmente permanece inalterada; somente os componentes que fazem interface com o hardware e o sistema operacional são modificados.
2. *Especialização de ambiente.* Versões da aplicação são criadas para lidar com ambientes operacionais e dispositivos periféricos específicos. Por exemplo, um sistema para serviços de emergência pode existir em versões diferentes dependendo do tipo de sistema de rádio usado. Nesse caso, os componentes de sistema são alterados para refletir a funcionalidade do equipamento de comunicação usado.
3. *Especialização funcional.* Versões da aplicação são criadas para clientes específicos com requisitos diferentes. Por exemplo, um sistema automatizado de biblioteca pode ser modificado dependendo se for usado em uma biblioteca pública, uma biblioteca de consulta ou uma biblioteca de universidade. Nesse caso, componentes que implementam a funcionalidade podem ser modificados e componentes novos podem ser adicionados ao sistema.
4. *Especialização de processo.* O sistema é adaptado para lidar com processos de negócio específicos. Por exemplo, um sistema de pedidos pode ser adaptado para lidar com um processo de pedidos centralizado em uma empresa e um processo distribuído em uma outra.

Linhas de produtos de software são projetadas para reconfiguração. Essa reconfiguração pode envolver a adição ou remoção de componentes do sistema, definição de parâmetros e restrições para os componentes, além da inclusão de conhecimento de processos de negócio. Linhas de produtos de software podem ser configuradas em dois pontos no processo de desenvolvimento:

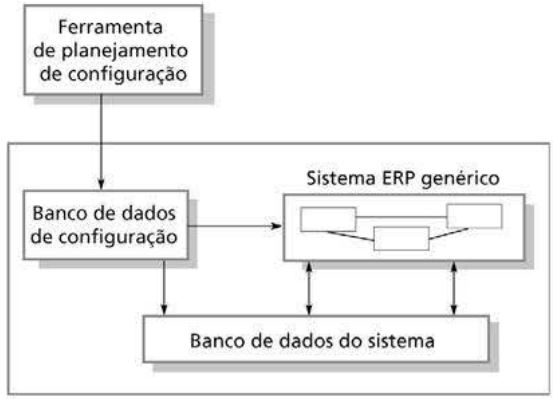
- *Configuração em tempo de implantação* na qual um sistema genérico é projetado por configuração por um cliente ou consultores que trabalham com o cliente. O conhecimento dos requisitos específicos do cliente e do ambiente operacional do sistema são incorporados em um conjunto de arquivos de configuração usados pelo sistema genérico.
- *Configuração em tempo de projeto* na qual a organização que está desenvolvendo o software modifica o núcleo comum da linha de produtos por desenvolvimento, seleção e adaptação de componentes para criar um novo sistema para o cliente.

A configuração em tempo de implantação é a abordagem usada nos pacotes de software verticais que são projetados para uma aplicação específica tal como um sistema de gerenciamento de informações hospitalares. É também usada em sistemas do tipo ERP (*Enterprise Resource Planning*) (O'Leary, 2000) tal como os produzidos pela empresas SAP e BEA. Estes são sistemas grandes e integrados projetados para apoiar processos de negócios, tais como pedidos e faturas, gerenciamento de estoques e cronogramas de manufatura. O processo de configuração desses sistemas envolve a obtenção de informações detalhadas sobre o negócio do cliente e os processos de negócios e, então, a inclusão dessas informações em um banco de dados de configuração. Isso requer, muitas vezes, conhecimento detalhado das notações e ferramentas de configuração, e é geralmente feito por consultores que trabalham junto com os clientes do sistema. A Figura 18.8 ilustra a organização de um sistema ERP.

O sistema genérico de ERP inclui uma grande quantidade de módulos que podem ser compostos de maneiras diferentes para criar um sistema específico. O processo de configuração envolve a escolha de quais módulos serão incluídos, a configuração desses módulos individuais, a definição de processos e regras de negócios e a definição da estrutura e organização do banco de dados de sistema.

Figura 18.8

Configuração de um sistema ERP.



Os sistemas ERP são, talvez, o exemplo mais difundido de reuso de software. A maioria das empresas usa esses sistemas para apoiar alguma ou todas as suas funções. Contudo, há a limitação óbvia da funcionalidade do sistema ao núcleo genérico. Além disso, processos e operações da empresa precisam ser expressos em uma linguagem de configuração de sistema, e pode haver uma diferença entre os conceitos no negócio e os conceitos apoiados na linguagem de configuração. Por exemplo, em um sistema ERP vendido para uma universidade, o conceito de um cliente tinha que ser definido. Isso causou problemas reais porque as universidades têm vários tipos de cliente (estudantes, órgãos de apoio à pesquisa, bolsas de estudo etc.) e nenhum desses são comparáveis a um cliente comercial. A grande diferença entre o modelo de negócio usado pelo sistema e o usado pelo cliente faz com que seja altamente provável que o sistema ERP não atenda às reais necessidades do cliente (Scott, 1999).

A abordagem alternativa ao reuso de família de aplicações é a configuração pelo fornecedor do sistema antes da entrega para o cliente. O fornecedor começa com um sistema genérico e então, por meio da modificação e ampliação dos módulos do sistema, cria um sistema específico que fornece a funcionalidade exigida pelo cliente. Essa abordagem geralmente envolve alteração e ampliação do código-fonte do núcleo do sistema e, assim, é possível maior flexibilidade do que com a configuração em tempo de implantação.

Linhas de produtos de software geralmente surgem de aplicações existentes. Ou seja, uma organização desenvolve uma aplicação e, quando uma nova aplicação é necessária, a aplicação existente é usada como base para a nova aplicação. Demandas adicionais por novas aplicações provocam a continuidade do processo. Contudo, como a mudança tende a corromper a estrutura da aplicação, em algum estágio deve ser tomada uma decisão específica para projetar uma linha de produtos genérica. Esse projeto baseia-se no reuso do conhecimento adquirido com o desenvolvimento do conjunto inicial de aplicações.

Você pode pensar em linhas de produtos de software como instâncias e especializações de arquiteturas de aplicações mais genéricas, conforme explicado no Capítulo 13. Uma arquitetura de aplicação é muito geral; linhas de produtos de software especializam a arquitetura para um tipo específico de aplicação. Por exemplo, considere um sistema de linha de produtos projetado para lidar com o envio de veículos para serviços de emergência. Os operadores desse sistema anotam as chamadas sobre os incidentes, encontram o veículo apropriado e o enviam ao local do incidente. Os desenvolvedores podem comercializar as versões desse sistema para os serviços de polícia, bombeiros e ambulâncias.

Esse sistema de envio de veículos é um exemplo de um sistema de gerenciamento de recursos cuja arquitetura de aplicação é mostrada na Figura 18.9. Você pode verificar como essa estrutura de quatro comandos é instanciada na Figura 18.10, que mostra os módulos que poderiam ser incluídos em uma linha de produtos de sistema de envio de veículos. Os componentes de cada nível do sistema da linha de produtos são:

1. No nível da interface com o usuário, há componentes que fornecem uma interface de display de operador e uma interface com os sistemas de comunicações usados.
2. No nível de gerenciamento de I/O (nível 2), há componentes que lidam com a autenticação do operador, geram relatórios de incidentes e veículos enviados, apóiam a saída de mapas e planejamento de rotas e fornecem um mecanismo para que os operadores consultem os bancos de dados do sistema.
3. No nível de gerenciamento de recursos (nível 3), há componentes que permitem aos veículos serem localizados e enviados, componentes para atualizar o estado dos veículos e equipamentos e um componente para registrar detalhes de incidentes.

Figura 18.9

Arquitetura de um sistema de alocação de recursos.

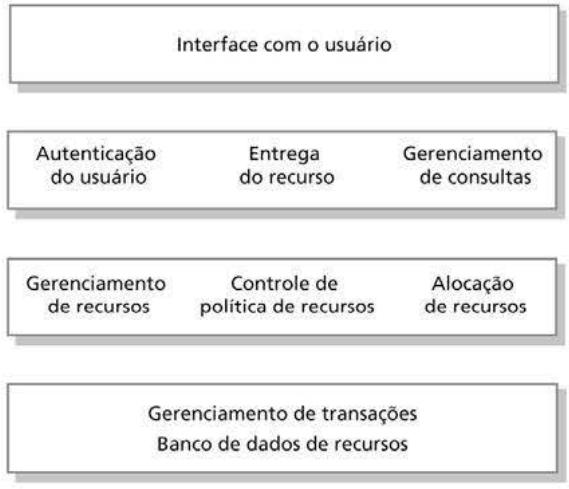
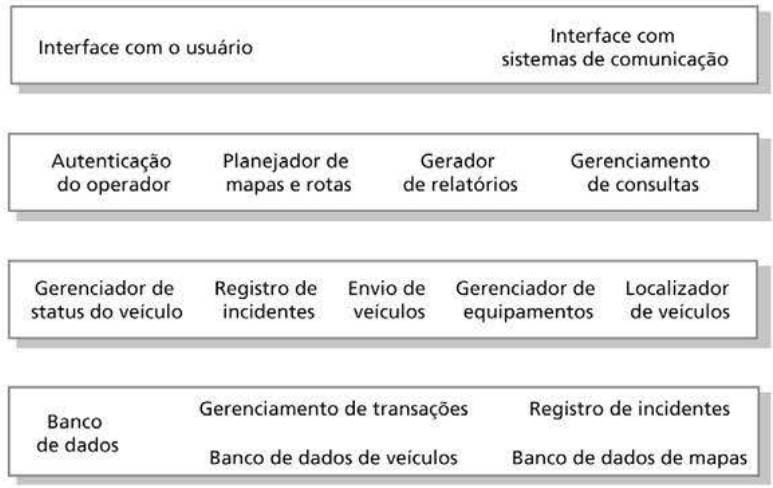


Figura 18.10

Arquitetura de linha de produtos de um sistema de envio de veículos.



4. No nível de base de dados, bem como no apoio geral de gerenciamento de transações, há bases de dados separadas dos veículos, equipamentos e mapas.

Para criar uma versão específica desse sistema, você pode ter que modificar componentes individuais. Por exemplo, a polícia tem uma grande quantidade de veículos, mas um pequeno número de tipos de veículos; por outro lado o serviço de bombeiros tem mais tipos de veículos especializados, assim, pode ser necessário incorporar um estrutura diferente de banco de dados de veículos ao sistema.

A Figura 18.11 mostra as etapas envolvidas na adaptação de uma família de aplicações para criar uma nova aplicação. As etapas nesse processo geral são:

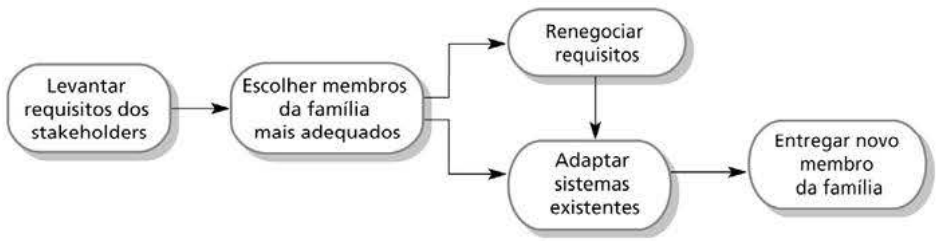
1. *Levantar requisitos dos stakeholders.* Você pode começar com um processo normal de engenharia de requisitos. Contudo, pelo fato de o sistema já existir, você precisará demonstrar e ter os experimentos dos stakeholders com esse sistema, expressando seus requisitos, como as modificações das funções fornecidas.
2. *Escolher um membro da família mais adequado.* Os requisitos são analisados e o membro da família mais adequado é escolhido para a modificação. Este não precisa ser o sistema que foi demonstrado.
3. *Renegociar requisitos.* À medida que mais detalhes de mudanças necessárias surgem e o projeto é planejado, pode haver alguma renegociação de requisitos para minimizar as alterações necessárias.
4. *Adaptar o sistema existente.* Novos módulos são desenvolvidos para o sistema existente e seus módulos são adaptados para atender aos novos requisitos.
5. *Entregar novo membro de família.* A nova instância da linha de produtos é entregue ao cliente. Neste estágio, você deve documentar suas características principais que podem ser usadas como base para desenvolvimentos futuros do sistema.

Quando você cria um novo membro de uma família de aplicações, você pode ter de assumir um compromisso entre reusar tantas aplicações genéricas quanto possível e satisfazer os requisitos detalhados do stakeholder. Quanto mais detalhados os requisitos de sistema, menos provável será que os componentes existentes atendam a esses requisitos. Entretanto, se os envolvidos puderem ser flexíveis e limitar as modificações necessárias do sistema, você poderá entregar o sistema mais rapidamente e a um custo menor.

Em geral, o desenvolvimento de aplicações pela adaptação de uma versão genérica da aplicação significa que uma proporção muito alta de códigos de aplicação é reusada. Além disso, a experiência com a aplicação é muitas vezes transferível

Figura 18.11

Desenvolvimento de instância de produto.



de um sistema para um outro, de tal forma que, quando os engenheiros de software se juntam à equipe de desenvolvimento, seu processo de aprendizado é abreviado. O teste é simplificado porque os testes de grandes partes da aplicação também podem ser reusados, diminuindo o tempo total de desenvolvimento da aplicação.

PONTOS-CHAVE

- As vantagens de reuso de software são custos menores, desenvolvimento de software mais rápido e riscos baixos. A confiança no sistema é aumentada e os especialistas podem ser usados mais eficientemente pela concentração de seu conhecimento no projeto de componentes reusáveis.
- *Design patterns* são abstrações de alto nível que documentam soluções bem-sucedidas de projeto. Eles são fundamentais para o reuso de projeto no desenvolvimento orientado a objetos. Uma descrição de um padrão deve incluir o nome, uma descrição do problema e da solução, além de uma declaração dos resultados e compromissos para o uso do padrão.
- Geradores de programa são uma abordagem alternativa ao *concept reuse*, no qual os conceitos reusáveis são incorporados a um sistema gerador. O projetista especifica as abstrações necessárias usando uma linguagem específica de domínio, e um programa executável é gerado.
- Frameworks de aplicação são conjuntos de objetos concretos e abstratos projetados para serem reusados através de especialização e adição de novos objetos.
- O reuso de produtos COTS é dedicado ao reuso de grandes sistemas comerciais. Eles fornecem uma grande quantidade de funcionalidade e seu reuso pode diminuir radicalmente os custos e o tempo de desenvolvimento.
- Problemas potenciais com o reuso baseado em COTS incluem a falta de controle sobre a funcionalidade e o desempenho, falta de controle sobre a evolução do sistema, a necessidade de suporte de fornecedores externos e as dificuldades em assegurar a interoperabilidade dos sistemas.
- Sistemas ERP (*Enterprise Resource Planning*) são largamente usados. Sistemas ERP específicos são criados pela configuração de um sistema genérico ao tempo de implantação com informações sobre o negócio do cliente.
- Linhas de produtos de software são aplicações relacionadas desenvolvidas a partir de uma ou mais aplicações-base. Um sistema genérico é adaptado e especializado para atender aos requisitos específicos de funcionalidade, plataforma-alvo ou configuração operacional.

LEITURAS SUGERIDAS

Reuse-based software engineering. Uma explicação abrangente de abordagens diferentes para reuso de software. Os autores abordam aspectos de reuso técnico e processos de gerenciamento de reuso. (H. Mili, et al., 2002, John Wiley & Sons.)

"A lifecycle process for the effective reuse of commercial off-the-shelf software". Uma boa introdução geral, que abrange as vantagens e as desvantagens no uso de COTS na engenharia de software. (C. L. Braun, *Proc. Symposium on Software Reusability*, Los Angeles, 1999. ACM Press. Disponível na ACM Digital Library.)

Design patterns: elements of reusable object-oriented software. Este é o manual original de padrões de software que apresenta os padrões a uma ampla comunidade. (E. Gamma, et al., 1995, Addison-Wesley.)

"Aspect-oriented programming". Esta edição especial da CACM tem um número de artigos sobre desenvolvimento de software orientado a aspectos. É um excelente ponto de partida para leitura sobre este tópico. (*Comm. ACM*, 44 (10), out. 2001.)

EXERCÍCIOS

18.1 Quais são os principais fatores técnicos e não técnicos que impedem o reuso de software? Você reusa muito software? Se não, por quê?

18.2 Sugira por que as reduções de custos no reuso de software existente não são simplesmente proporcionais ao tamanho dos componentes reusados.

18.3 Forneça quatro circunstâncias nas quais você poderia não recomendar o reuso de software.

18.4 Por que os padrões são uma maneira efetiva de reuso de projeto? Quais são as desvantagens dessa abordagem de reuso?

18.5 Além dos domínios de aplicação explicados aqui, sugira dois outros domínios nos quais o reuso baseado em geradores poderia ser bem-sucedido. Explique por que essa abordagem de reuso terá custo adequado nesses domínios.

18.6 Explique por que os adaptadores geralmente são necessários quando os sistemas são construídos pela integração de produtos COTS.

18.7 Identifique seis possíveis riscos que podem surgir quando os sistemas são construídos usando COTS. Quais etapas uma empresa pode seguir para reduzir esses riscos?

18.8 Usando uma arquitetura geral de informações de sistema (explicada no Capítulo 13) como ponto de partida, projete uma família de aplicações de sistemas de informações de biblioteca que poderia ser usada em acervos de livros, filmes, músicas e recortes de jornal.

18.9 Usando o exemplo do sistema de estação de tempo descrito no Capítulo 14, sugira uma arquitetura para uma família de aplicações dedicadas à monitoração e coleta de dados remotos.

18.10 O reuso de software levanta um número de aspectos de direitos autorais e propriedade intelectual. Se um cliente paga a uma empresa software para desenvolver um sistema, quem tem o direito de reuso do código desenvolvido? A empresa de software tem o direito de usar esse código como base de um componente genérico? Quais mecanismos de pagamento poderiam ser usados para reembolsar os fornecedores dos componentes reusáveis? Explique esses e outros aspectos éticos associados ao reuso de software.

19

Engenharia de software baseada em componentes

Objetivos

O objetivo deste capítulo é descrever um processo de desenvolvimento de software baseado na composição de componentes reusáveis padronizados. Após ler este capítulo, você:

- saberá que a engenharia de software baseada em componentes concentra-se no desenvolvimento de componentes padronizados com base em um modelo de componente e na composição desses componentes em sistemas de aplicação;
- compreenderá qual é o significado de um componente e um modelo de componente;
- conhecerá as atividades principais no processo CBSE e compreenderá por que é necessário comprometer-se com os requisitos de maneira que os componentes possam ser reusados;
- compreenderá algumas das dificuldades e problemas que surgem durante o processo de composição de componentes.

Conteúdo

19.1 Componentes e modelos de componentes
19.2 Processo CBSE
19.3 Composição de componentes

Conforme sugeri no Capítulo 18, a engenharia de software baseada em reuso está se tornando a principal abordagem de desenvolvimento para sistemas corporativos e comerciais. As entidades reusadas variam de pequenas funções a sistemas inteiros de aplicação. Contudo, até recentemente, era difícil reusar componentes médios de programas. Componentes médios são significativamente maiores do que objetos ou procedimentos individuais, com mais funcionalidade, mas são menores e mais específicos do que sistemas de aplicação. Felizmente, os avanços em padronização promovidos pelos principais fornecedores de software possibilitaram a interoperação de componentes dentro de um framework, como CORBA. Isso criou oportunidades para reuso sistemático por meio de engenharia de software baseada em componentes.

A engenharia de software baseada em componentes (CBSE — Component-Based Software Engineering) surgiu final da década de 1990 como uma abordagem baseada em reuso para desenvolvimento de sistemas de software. Sua criação foi motivada pela frustração dos projetistas quanto

ao desenvolvimento orientado a objetos não conduzir ao reuso amplo, como originalmente fora sugerido. Classes de objetos individuais eram muito detalhadas e específicas e, muitas vezes, tinham de ser ligadas a uma aplicação no momento da compilação. Era necessário conhecer detalhadamente as classes e usá-las, o que significava geralmente que era necessário possuir o código-fonte do componente. Isso tornou difícil propagar a idéia de que os objetos eram componentes reusáveis. Apesar das previsões iniciais otimistas, nenhum mercado significativo para objetos individuais se desenvolveu.

O CBSE é um processo de definição, implementação e integração ou composição de componentes independentes não firmemente acoplados ao sistema. O CBSE tornou-se uma importante abordagem de desenvolvimento de software, pois os sistemas de software estão se tornando maiores e complexos e os clientes exigem software mais confiável e que seja desenvolvido mais rapidamente. A única maneira de lidar com a complexidade e entregar um software melhor mais rapidamente é reusar componentes de software mais do que reimplementá-los.

Os pontos essenciais da engenharia de software baseada em componentes são:

1. *Componentes independentes* completamente especificados por suas interfaces. Deve haver uma clara separação entre a interface do componente e sua implementação de maneira que a implementação de um componente possa ser substituída por uma outra sem mudança do sistema.
2. *Padrões de componentes* que facilitam a integração de componentes. Esses padrões são incorporados em um modelo de componente e definem, no mínimo, como as interfaces de componentes devem ser especificadas e como os componentes devem se comunicar. Alguns modelos definem interfaces que devem ser implementadas por todos os componentes em conformidade com os padrões. Se os componentes estiverem em conformidade com os padrões, a operação será independente de sua linguagem de programação. Componentes escritos em linguagens diferentes podem ser integrados em um mesmo sistema.
3. *Middleware* que fornece apoio de software para integração de componentes. Para fazer com que componentes independentes e distribuídos trabalhem juntos, é necessário o apoio de um middleware que manipule as comunicações entre componentes. Um middleware como CORBA (Pope, 1998), explicado no Capítulo 12, manipula recursos de baixo nível eficientemente e permite enfocar problemas relacionados à aplicação. Além disto, o middleware para implementar um modelo de componente pode fornecer apoio à alocação de recursos, gerenciamento de transações, proteção e concorrência.
4. *Processo de desenvolvimento* voltado à engenharia de software baseada em componentes. Ao tentar adicionar uma abordagem baseada em componentes em um processo de desenvolvimento voltado à produção de software original, você descobrirá que suposições inerentes no processo limitam o potencial do CBSE. Explico os processos de desenvolvimento CBSE na Seção 19.2.

O desenvolvimento baseado em componentes está sendo cada vez mais adotado como abordagem principal para engenharia de software mesmo que os componentes reusáveis não estejam disponíveis. As bases do CBSE são princípios seguros de projeto que se apóiam na construção de software compreensível e fácil de manter. Os componentes são independentes, portanto não interferem na operação uns dos outros. Detalhes de implementação são ocultados, de maneira que a implementação do componente pode ser mudada sem afetar o restante do sistema. Os componentes se comunicam por interfaces bem definidas; se essas interfaces passarem por manutenção, um componente poderá ser substituído por outro com funcionalidade adicional ou aprimorada. Além disso, infra-estruturas de componentes fornecem plataformas de alto nível que reduzem os custos de desenvolvimento da aplicação.

Embora a CBSE esteja se desenvolvendo rapidamente em uma abordagem principal para desenvolvimento de software, alguns problemas permanecem:

1. *Confiabilidade de componentes.* Os componentes são unidades 'caixa-preta' de programa, e o código-fonte do componente pode não estar disponível aos usuários. Nesses casos, como um usuário pode saber que um componente é confiável? O componente pode ter modos de falha não documentada que comprometem o sistema em que é usado. Seu comportamento não funcional pode não ser o esperado e, mais seriamente, o componente caixa-preta pode ser um cavalo de Tróia ocultando código maléfico que abre uma brecha na proteção do sistema.
2. *Certificação de componentes.* Estreitamente relacionada à confiabilidade está a questão da certificação. Foi proposto que avaliadores independentes devem certificar componentes para assegurar se estes são confiáveis. Contudo, não está claro como isso pode funcionar. Quem pagaria pela certificação, quem seria responsável se o componente não operar como certificado e como poderia o certificador limitar sua responsabilidade? Em minha opinião, a única solução viável é se certificar de que os componentes estejam em conformidade com uma especificação formal. Contudo, a indústria não parece interessada em pagar por isto.
3. *Previsão de propriedade emergente.* Conforme explicado no Capítulo 2, todos os sistemas têm propriedades emergentes e a tentativa de prever e controlar essas propriedades emergentes é importante no processo de desenvolvimento

do sistema. Devido aos componentes serem opacos, a previsão de suas propriedades emergentes é particularmente difícil. Conseqüentemente, você pode descobrir que, quando os componentes são integrados, o sistema resultante tem propriedades indesejáveis que limitam seu uso.

4. *Compromissos de requisitos.* Geralmente, você precisa ter compromissos entre requisitos ideais e componentes disponíveis no processo de especificação e projeto do sistema. Neste momento, ter esses compromissos é um processo intuitivo. Nós necessitamos de um método de análise estruturado e sistemático de compromissos para ajudar os projetistas a selecionar e configurar componentes.

O principal uso do CBSE até o momento foi construir sistemas de informações, como sistemas de e-commerce. Os componentes reusados são desenvolvidos internamente ou adquiridos de fornecedores conhecidos e confiáveis. Embora alguns fornecedores vendam componentes on-line, muitas empresas ainda relutam em confiar em componentes binários adquiridos externamente. É improvável que a visão completa do CBSE com fornecedores de componentes especializados seja percebida até que esses problemas maiores tenham sido resolvidos.

19.1 Componentes e modelos de componentes

Há um consenso geral na comunidade de que um componente é uma unidade de software independente que pode ser composta por outros componentes para criar um sistema de software. Além disso, pessoas diferentes propuseram definições de um componente de software. Councill e Heineman (Councill e Heineman, 2001) definem um *componente* como:

um elemento de software que está em conformidade com um modelo de componente e pode ser independentemente implantado e composto sem modificação de acordo com um padrão de composição.

Essa definição é essencialmente baseada em padrões — uma unidade de software que esteja em conformidade com esses padrões é um componente. Szyperski (Szyperski, 2002), contudo, não menciona padrões em sua definição de componente, mas enfoca, em vez disso, características principais de componentes:

Um componente de software é uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas somente. Um componente de software pode ser implantado independentemente e está sujeito à composição por terceiros.

Szyperski também define que um componente não tem estado externamente observável. Isso significa que cópias dos componentes são indistinguíveis. Contudo, alguns modelos de componente, como o modelo Enterprise Java Beans, permitem componentes com estado e, assim, claramente não estão de acordo com a definição de componente de Szyperski. Enquanto componentes sem estado são certamente mais simples de usar, penso que a CBSE deve acomodar ambos, componentes com e sem estado.

O que essas definições têm em comum é que elas concordam que componentes são independentes e que são as unidades fundamentais de composição em um sistema. Em minha opinião, uma definição completa de um componente pode ser derivada de ambas as propostas. A Tabela 19.1 mostra o que considero ser as características essenciais de um componente baseado em CBSE.

Essas definições formais de componente são um tanto abstratas e não fornecem realmente um quadro claro do que um componente faz. Uma das maneiras mais úteis de se considerar um componente é como um provedor de serviços independente. Quando um sistema necessita de algum serviço, chama um componente para fornecer esse serviço sem se preocupar com onde esse componente está sendo executado ou a linguagem de programação usada para desenvolver o componente. Por exemplo, um componente em um sistema de biblioteca pode fornecer um serviço de busca que permite aos usuários buscar catálogos diferentes; um componente que faz a conversão de um formato gráfico em outro (por exemplo, TIFF em JPEG) fornece um serviço de conversão de dados.

Observando um componente como um provedor de serviços, duas características críticas de um componente reusável são enfatizadas:

1. O componente é uma entidade executável independente. O código-fonte não está disponível, de maneira que o componente não precisa ser compilado antes de ser usado com outros componentes do sistema.
2. Os serviços oferecidos por um componente são disponibilizados por meio de uma interface, e todas as interações ocorrem por essa interface. A interface com o componente é expressa em termos de operações parametrizadas e seu estado interno nunca é exposto.

Os componentes são definidos por suas interfaces e, na maioria dos casos, podem ser imaginados como se tivessem duas interfaces relacionadas, como mostrado na Figura 19.1.

Tabela 19.1 Características de componente

Plano	Descrição
Padronizado	A padronização de componentes significa que um componente usado em um processo CBSE precisa estar de acordo com algum modelo padronizado de componente. Esse modelo pode definir interfaces de componente, metadados de componente, documentação, composição e implantação.
Independente	Um componente deve ser independente — deve ser possível compô-lo e implantá-lo sem precisar usar outros componentes específicos. Em situações em que o componente necessita de serviços fornecidos externamente, estes devem ser explicitamente definidos em uma especificação de interface '*requires*'.
Passível de composição	Para um componente poder ser composto, todas as interações externas devem ocorrer por meio de interfaces publicamente definidas. Além disso, deve fornecer acesso externo às informações sobre si próprio, como seus métodos e atributos.
Implantável	Para ser implantável, um componente precisa ser autocontido e deve ser capaz de operar como uma entidade independente sobre uma plataforma de componente que implementa o modelo de componente. Isso geralmente significa que o componente é binário e não precisa ser compilado antes de ser implantado.
Documentado	Componentes precisam ser completamente documentados de maneira que os usuários possam decidir se eles atendem ou não às suas necessidades. A sintaxe e, idealmente, a semântica de todas as interfaces de componente precisam ser especificadas.

Figura 19.1

Interfaces de componente.

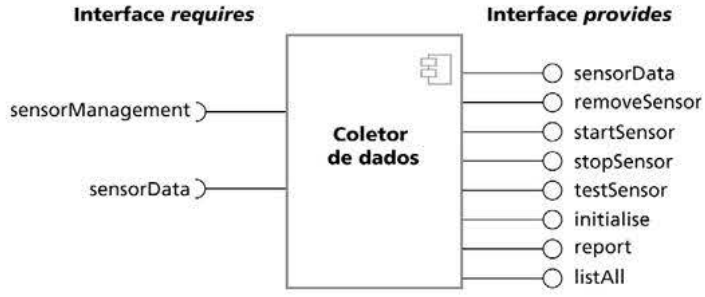
1. Uma interface *provides* define os serviços fornecidos pelo componente. Essa interface, essencialmente, é o componente API. Ele define os métodos que podem ser chamados pelo usuário do componente. As interfaces *provides* são indicadas por um círculo no fim de uma linha do ícone de componente.
2. Uma interface *requires* especifica quais serviços devem ser fornecidos pelos outros componentes no sistema. Se estes não estiverem disponíveis, o componente não funcionará. Isso não compromete a independência ou a facilidade de implantação do componente porque não é necessário que um componente específico seja usado para fornecer os serviços. As interfaces *requires* são indicadas por um semicírculo no fim de uma linha do ícone de componente. Observe que os ícones de interfaces *provides* e *requires* encaixam-se um no outro.

Por exemplo, a Figura 19.2 mostra um modelo de componente projetado para coletar e comparar cuidadosamente as informações de um conjunto de sensores. Ele realiza as coletas de dados autonomamente a cada período de tempo e, sob requisição, fornece dados comparados para um componente requisitante. A interface *provides* inclui os métodos *add, remove, start, stop* e *test* de sensores. Também inclui métodos para emissão de relatórios (**report** e **listAll**) que reportam os dados coletados e a configuração de sensores. Embora não tenha sido mostrado aqui, esses métodos associam naturalmente parâmetros de especificação de localizações de sensores e assim por diante.

O componente coletor requer que os sensores forneçam uma interface de gerenciamento e interface de dados. Esses sensores têm parâmetros que especificam a operação e os dados a serem coletados. Eu projetei deliberadamente a interface necessária de modo a não incluir operações específicas como **Test**. A interface *requires* mais abstrata permite que o com-

Figura 19.2

Modelo de componente coletor de dados.



ponente coletor seja usado em sensores com interfaces diferentes. Um componente adaptador é usado como uma interface entre o coletor e a interface de sensor de hardware específico.

As classes de objetos associaram métodos claramente similares aos métodos definidos nas interfaces de componente. Qual é, então, a distinção entre componentes e objetos? Componentes são geralmente desenvolvidos por meio de uma abordagem orientada a objetos, mas eles diferem de objetos de várias maneiras maneiras importantes:

1. *Componentes são entidades implantáveis.* Isto é, eles não são compilados em um programa de aplicação, mas sim instalados diretamente sobre uma plataforma de execução. Os métodos e atributos definidos em suas interfaces podem, então, ser acessados por outros componentes.
2. *Componentes não definem tipos.* Uma definição de classe define um tipo abstrato de dados e os objetos são instâncias desse tipo. Um componente é uma instância, não um template usado para definir uma instância.
3. *Implementações de componentes são opacas.* Os componentes são, em princípio pelo menos, completamente definidos pela especificação de interface. A implementação é ocultada dos usuários do componente. Os componentes são freqüentemente entregues como unidades binárias e, assim, o comprador do componente não tem acesso à implementação.
4. *Componentes são independentes de linguagem.* As classes de objeto precisam seguir as regras de uma linguagem de programação orientada a objetos e, em geral, podem somente interoperar com outras classes naquela linguagem. Embora os componentes sejam geralmente implementados por meio de uma linguagem orientada a objetos, como Java, você pode implementá-los em linguagens de programação não orientadas a objetos.
5. *Componentes são padronizados.* Diferente das classes de objeto que podem ser implementadas de alguma maneira, os componentes devem estar de acordo com algum modelo de componente que restringe sua implementação.

19.1.1 Modelos de componentes

Um modelo de componente é uma definição de padrões para implementação, documentação e implantação de componentes. Esses padrões são para que os desenvolvedores se assegurem de que os componentes podem operar entre si. Eles se destinam também aos fornecedores de infra-estruturas de execução de componentes que fornecem middleware para apoiar a operação de componentes. Muitos modelos de componentes foram propostos, mas os modelos mais importantes são o CORBA da OMG, o Enterprise Java Bens da Sun e o COM+ da Microsoft (Blevins, 2001; Ewald, 2001; Wang, et al., 2001).

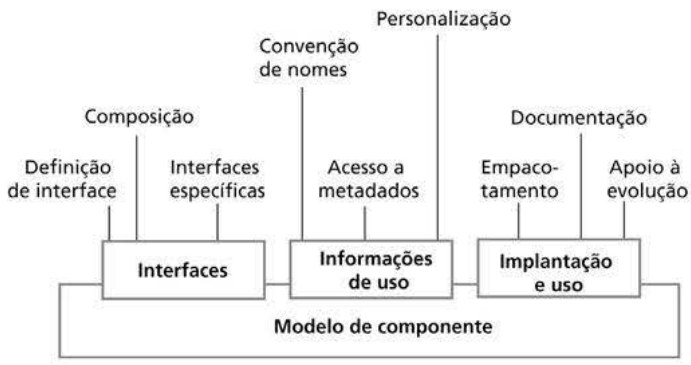
As tecnologias específicas de infra-estrutura, como COM+ e EJB usadas em CBSE, são muito complexas. Conseqüentemente, é difícil descrever essas tecnologias sem entrar em uma grande quantidade de detalhes de implementação sobre as suposições que cada abordagem contém e as interfaces usadas. Em vez de me aprofundar nesses detalhes, enfoco os elementos fundamentais dos modelos de componentes.

Os elementos básicos de um modelo ideal de componente são explicados por Weinreich e Sametinger (Weinreich e Sametinger, 2001). Faço um resumo desses elementos de modelo na Figura 19.3. Esse diagrama mostra que os elementos em um modelo de componente podem ser classificados como elementos relacionados às interfaces de componente, elementos relacionados a informações de que você necessita para usar o componente em um programa e elementos relacionados à implantação de componentes.

Os elementos de definição de um componente são suas interfaces. O modelo de componente especifica como as interfaces devem ser definidas e os elementos, como nomes de operação, parâmetros e exceções, que devem ser incluídos em uma definição de interface. O modelo deve também especificar a linguagem usada para definir as interfaces (o IDL). No

Figura 19.3

Elementos básicos de um modelo de componente.



CORBA e no COM+, essa é uma linguagem específica de definição de interface; EJB é específica para Java de modo que é usada como o IDL. Alguns modelos de componente requerem interfaces específicas que devem ser definidas por um componente. Elas são usadas para compor o componente com a infra-estrutura de modelo de componente que fornece serviços padronizados, como gerenciamento de proteção e de transação.

Para que os componentes sejam distribuídos e acessados remotamente, eles necessitam ter um nome exclusivo ou manipulador associado a eles. No COM+, isto é um identificador exclusivo de 128 bits. No modelo de componente CORBA e no EJB, é um nome hierárquico com a raiz baseada em um nome de domínio de Internet. Os metadados de componente são dados sobre o componente em si, como as informações sobre suas interfaces e atributos. Os metadados são importantes, pois os usuários do componente podem encontrar quais serviços são fornecidos e necessários. Implementações de modelo de componente incluem normalmente maneiras específicas (como o uso de uma interface de reflexão em Java) para acessar os metadados de componente.

Componentes são entidades genéricas e, quando implantados, precisam ser configurados para seu ambiente de aplicação específico. Por exemplo, o componente **Data collector** mostrado na Figura 19.2 pode ser configurado com o número máximo em um conjunto de sensores. O modelo de componente deve, portanto, especificar como os componentes binários podem ser configurados para um ambiente de implantação.

Uma parte importante de um modelo de componente é uma definição de como os componentes devem ser empacotados para implantação como entidades executáveis independentes. Devido aos componentes serem entidades independentes, eles precisam ser empacotados com qualquer coisa que não é fornecida pela infra-estrutura de componente ou não é definida em uma interface *requires*. As informações de implantação incluem informações sobre os conteúdos de um pacote e sua organização binária.

Inevitavelmente, conforme novos requisitos surgem, os componentes devem ser mudados ou realocados. O modelo de componente deve, portanto, incluir regras que definam quando e como a substituição de componentes é permitida. Finalmente, o modelo de componente deve definir a documentação de componente a ser produzida. Isso é usado para encontrar o componente e decidir se ele é apropriado.

Modelos de componente não são somente padrões; eles são também a base para o middleware do sistema que fornece apoio aos componentes em execução. Weinreich e Sametinger (Weinreich e Sametinger, 2001) usam a analogia de um sistema operacional para explicar modelos de componentes. Um sistema operacional fornece um conjunto de serviços genéricos que pode ser usado pelas aplicações. Uma implementação de modelo de componente fornece serviços compartilhados comparáveis aos dos componentes. A Figura 19.4 mostra alguns dos serviços que podem ser fornecidos pela implementação de um modelo de componente.

Os serviços fornecidos pela implementação de modelo de componente recaem em duas categorias:

1. *Serviços de plataforma.* Esses serviços fundamentais permitem que os componentes se comuniquem uns com os outros. CORBA é um exemplo de plataforma de modelo de componente. Os serviços de plataforma foram descritos no Capítulo 12.
2. *Serviços horizontais.* Esses serviços independentes de aplicação serão provavelmente usados por muitos componentes diferentes. A disponibilidade dos serviços reduz os custos de desenvolvimento de componente e significa que incompatibilidades potenciais entre componentes podem ser evitadas.

Para fazer uso desses serviços fornecidos por uma infra-estrutura de modelo de componente, os componentes são implantados em um contêiner predefinido e padronizado. Um *contêiner* é um conjunto de interfaces usado para acessar as implementações dos serviços de apoio. Incluir o componente em um contêiner fornece automaticamente acesso aos serviços. As próprias interfaces de componente não são acessadas diretamente por outros componentes, mas sim por meio de contêineres.

Figura 19.4

Serviços fornecidos por um modelo de componente.

19.1.2 Desenvolvimento de componente para reuso

A visão de longo prazo do CBSE é que haverá fornecedores de componentes cujo negócio será baseado no desenvolvimento e na venda de componentes reusáveis. Conforme já disse, os problemas de confiança significam que um mercado aberto para componentes ainda não foi desenvolvido, e a maioria dos componentes reusados é desenvolvida dentro das empresas. Os componentes reusáveis não são desenvolvidos especialmente, mas baseiam-se em componentes existentes já implementados e usados em sistemas de aplicação.

Geralmente, componentes desenvolvidos internamente não são imediatamente reusáveis. Eles incluem características e interfaces específicas de aplicação improváveis de serem necessárias em outras aplicações. Portanto, você precisa adaptar, ampliar esses componentes e criar uma versão mais genérica e, portanto, mais reusável. Obviamente, isso tem um custo associado. Você precisa decidir primeiro se um componente tem possibilidade de ser reusado e, depois, se a redução de custo com o reuso justifica o custo de tornar o componente reusável.

Para responder à primeira dessas perguntas, é necessário decidir se o componente implementa uma ou mais abstrações estáveis de domínio. Abstrações estáveis de domínio são conceitos fundamentais no domínio da aplicação que mudam lentamente. Por exemplo, em um sistema bancário, as abstrações de domínio podem incluir contas, correntistas e extratos. Em um sistema de gerenciamento hospitalar, abstrações de domínio podem incluir pacientes, tratamentos e enfermeiras. Essas abstrações de domínio são algumas vezes chamadas de objetos de negócio. Se o componente for uma implementação de um objeto de negócio ou grupo de objetos comumente usado, ele provavelmente pode ser reusado.

Para responder à pergunta sobre a adequação de custo, você precisa avaliar os custos de mudanças necessárias para tornar o componente reusável. Esses custos são os custos de documentação de componente, de validação de componente e de tornar o componente mais genérico. As mudanças a serem feitas para tornar um componente mais reusável incluem:

- Remover métodos específicos de aplicação
- Mudar os nomes para torná-lo mais geral
- Adicionar métodos para obter uma abrangência funcional maior
- Tratar exceções consistentemente em todos os métodos
- Adicionar uma interface de 'configuração' para permitir que o componente seja adaptado a diferentes situações de uso
- Integrar componentes necessários para aumentar a independência

O problema de tratamento de exceções é particularmente difícil. Em princípio, todas as exceções devem ser partes da interface do componente. Os componentes não devem tratar as exceções por si mesmos, pois cada aplicação terá seus próprios requisitos para tratamento de exceções. Antes, o componente deve definir quais exceções podem surgir e publicá-las como parte da interface. Por exemplo, um componente simples que implementa uma estrutura de pilha deve detectar e publicar as exceções de *overflow* e *underflow* da pilha. Na prática, contudo, um componente pode fornecer algum tratamento local de exceções, e mudar isso pode ter implicações sérias na funcionalidade do componente.

Mili et al. (Mili, et al., 2002) explicam maneiras de estimar os custos de tornar um componente reusável e o retorno sobre o investimento. Os benefícios do reuso, em vez de desenvolver novamente um componente, não são simplesmente ganhos de produtividade. Eles também incluem ganhos de qualidade, pois um componente reusado deve ser mais confiável, além de ganhos de *time-to-market*. Esses são os retornos obtidos com a implantação mais rápida de software. Mili et al. apresentam várias fórmulas para estimar esses ganhos, como no modelo COCOMO explicado no Capítulo 26 (Boehm, et al., 2000). Contudo, os parâmetros dessas fórmulas são difíceis de identificar com precisão, e as fórmulas devem ser adaptadas às circunstâncias locais. Suspeito que esses fatores fazem com que muito poucos gerentes de projeto de software queiram confiar neles.

Obviamente, o fato de um componente ser reusável depende de seu domínio de aplicação e funcionalidade. Quando um componente torna-se mais genérico, sua reusabilidade é ampliada. Contudo, isso normalmente significa que o componente tem mais operações e é mais complexo, o que o torna mais difícil de compreender e usar.

Há um inevitável compromisso entre a reusabilidade e a usabilidade de um componente. Tornar o componente reusável envolve fornecer um conjunto de interfaces genéricas com operações que fornecem todas as maneiras nas quais o componente poderia ser usado. Tornar um componente usável significa fornecer uma interface mínima e simples que seja fácil de compreender. A reusabilidade adiciona complexidade e, por isso, reduz a facilidade de compreensão do componente. É, portanto, mais difícil decidir quando e como reusar esse componente. Ao projetar um componente reusável, você deve manter um compromisso entre a generalidade e facilidade de compreensão.

Outra fonte importante de componentes são os sistemas legados existentes. Conforme explicado no Capítulo 2, esses são sistemas que preenchem uma função importante de negócio, mas são escritos com tecnologias de software obsoletas. Devido a esse fato, pode ser difícil usá-los com novos sistemas. Contudo, se você converter esses velhos sistemas em componentes, sua funcionalidade pode ser reusada em novas aplicações.

Naturalmente, esses sistemas legados não têm interfaces *requires* e *provides* claramente definidas. Para tornar esses componentes reusáveis, você precisa passar por um *wrapper* que define as interfaces do componente. O wrapper oculta a complexidade inerente ao código e fornece uma interface para componentes externos para acessar os serviços fornecidos. Naturalmente esse wrapper é um fragmento de software relativamente complexo que precisa acessar a funcionalidade do sistema legado. Contudo, o custo de desenvolvimento do wrapper é, muitas vezes, menor do que o custo de reimplementação do sistema legado.

19.2 Processo CBSE

Sugeri, na introdução, que o reuso bem-sucedido de componentes requer um processo de desenvolvimento adaptado do CBSE. A estrutura de tal processo foi explicada no Capítulo 4; a Figura 19.5 mostra as principais subatividades dentro de um processo CBSE. Algumas das atividades dentro desse processo, como a descoberta inicial dos requisitos do usuário, são realizadas da mesma maneira que em outros processos de software. Contudo, as diferenças essenciais entre esse processo e o processo de software baseado no desenvolvimento original de software são:

1. Os requisitos do usuário são inicialmente desenvolvidos em linhas gerais antes de ser em detalhes, e os stakeholders são encorajados a serem tão flexíveis quanto possível na definição de seus requisitos. A razão para isso é que requisitos muito específicos limitam o número de componentes que podem atender a esses requisitos. Diferentemente do desenvolvimento incremental, contudo, é necessário um conjunto completo de requisitos de modo que você possa identificar tantos componentes quanto possíveis para reuso.
2. Requisitos são refinados e modificados no início do processo dependendo dos componentes disponíveis. Se os requisitos do usuário não puderem ser satisfeitos com base nos componentes disponíveis, você deve discutir os requisitos relacionados que podem ser apoiados. Os usuários podem mudar de idéia se isso significar a entrega mais barata ou mais rápida do sistema.
3. Há uma busca de componente adicional e uma atividade de refinamento de projeto depois que a arquitetura do sistema foi projetada. Alguns componentes aparentemente usáveis podem se tornar inadequados ou não funcionar adequadamente com outros componentes escolhidos. Embora não tenha sido mostrado na Figura 19.5, isso significa que mudanças adicionais de requisitos podem ser necessárias.
4. O desenvolvimento é um processo de composição no qual os componentes descobertos são integrados. Isso envolve a integração de componentes com a infra-estrutura de modelo de componente e, muitas vezes, desenvolvimento de 'código colante' para reconciliar as interfaces de componentes incompatíveis. Naturalmente, alguma funcionalidade adicional pode ser necessária junto com a fornecida pelos componentes usáveis. Assim, você deve desenvolver o processo de modo que os componentes possam ser reusados em sistemas futuros.

O estágio de projeto de arquitetura é particularmente importante. Durante o projeto de arquitetura, você pode finalmente decidir sobre um modelo de componente, embora, em muitos sistemas, essa decisão seja feita antes que a busca por componentes se inicie. Conforme explicado nos capítulos 11 a 13, você também estabelece a organização de alto nível do sistema e toma decisões sobre a distribuição e controle. Jacobsen et al. (Jacobsen, et al., 1997) descobriram que a definição de uma arquitetura robusta é crítica para o sucesso do reuso.

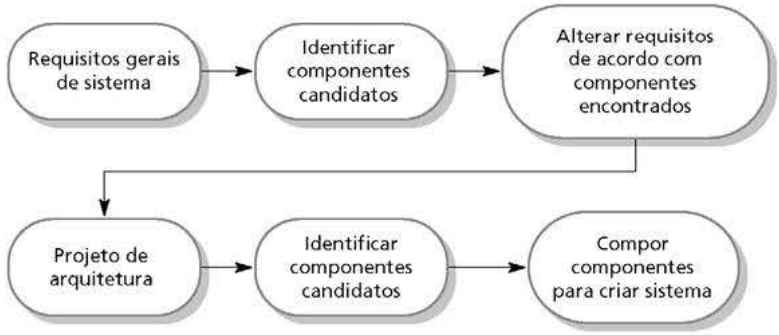
Uma atividade exclusiva do processo CBSE é a identificação de componentes. Isso envolve um número de subatividades, conforme mostrado na Figura 19.6. Há dois estágios no processo CBSE nos quais você precisa identificar componentes para uso possível no sistema. Nos estágios iniciais, o foco deve ser sobre a busca e a seleção. É necessário se certificar de que há componentes disponíveis que atendam a seus requisitos. Obviamente, deve ser feita alguma verificação inicial para analisar se o componente é adequado, mas pode ser que não sejam necessários testes detalhados. No estágio mais avançado, depois que a arquitetura do sistema foi projetada, você deve despender mais tempo na validação do componente. É preciso ter certeza de que os componentes identificados são realmente adequados para sua aplicação; caso não sejam, você precisará repetir os processos de busca e seleção.

O primeiro estágio na identificação de componentes é procurar por componentes disponíveis localmente ou de fornecedores confiáveis. A opinião de defensores do CBSE, como Szyperski (Szyperski, 2002), é que deve haver um mercado viável de componentes no qual os fornecedores externos disputem o fornecimento de componentes. Durante o período em que este livro estava sendo escrito, isso não aconteceu de nenhum modo significativo. A principal razão para isso é que os

Figura 19.5
Processo CBSE.

Figura 19.6

Processo de identificação de componente.



usuários de componentes externos estavam diante do risco de que esses componentes não funcionariam conforme anunciado. Se esse fosse o caso, os custos de reuso excederiam os benefícios, e poucos gerentes acreditariam que os riscos fossem aceitáveis. Outra importante razão por que os mercados de componentes não se desenvolveram é que muitos componentes estão em domínios especializados de aplicação. Não há um mercado suficientemente grande nesses domínios para que os fornecedores de componentes externos estabeleçam um negócio viável e de longo prazo.

Como conseqüência, a busca de componentes restringe-se muitas vezes a uma organização de desenvolvimento de software. Empresas de desenvolvimento de software podem criar seus próprios bancos de dados de componentes reusáveis sem os riscos inerentes ao uso de componentes de fornecedores externos.

Após a identificação de componentes candidatos pelo processo de busca, componentes específicos dessa lista precisam ser selecionados. Em alguns casos, isso é uma tarefa direta. Componentes da lista serão mapeados diretamente sobre os requisitos do usuário, e não haverá competição de componentes que atendam a esses requisitos. Em outros casos, contudo, o processo de seleção é muito mais complexo. Não haverá um mapeamento claro de requisitos para componentes, e você descobrirá que vários componentes precisam ser usados para atender a um requisito específico ou um grupo de requisitos. Infelizmente, é provável que diferentes requisitos exijam grupos diferentes de componentes, assim você precisa decidir quais composições de componentes fornecem a melhor cobertura de requisitos.

Após selecionar os componentes para possível inclusão em um sistema, você deve validá-los para verificar se eles se comportam conforme esperado. A extensão da validação depende da fonte dos componentes. Se você estiver usando um componente desenvolvido por uma fonte conhecida e confiável, o teste de componente separado pode ser desnecessário, havendo somente o teste de componente integrado a outros. Por outro lado, se você estiver usando um componente de uma fonte não conhecida, sempre verifique e teste esse componente antes de incluí-lo em seu sistema.

A validação de componente envolve o desenvolvimento de um conjunto de casos de teste para o componente (ou, possivelmente, a extensão de casos de teste fornecidos com o componente) e o desenvolvimento de um roteiro de teste para realizar os testes de componente. O principal problema com a validação de componente é que a especificação do componente pode não ser suficientemente detalhada para permitir o desenvolvimento de um conjunto completo de testes. Os componentes, em geral, são especificados informalmente, sendo a documentação formal somente sua especificação de interface. Pode não haver informações suficientes para o desenvolvimento de um conjunto completo de testes para comprovar que a interface do componente funcionará conforme esperado.

Um problema de validação adicional que pode surgir nesse estágio é que o componente pode ter características que interfiram em seu uso de componente. Geralmente, componentes reusáveis terão mais funcionalidades do que você necessita. Você pode simplesmente ignorar a funcionalidade não desejada, mas isso, às vezes, pode interferir em outros componentes ou no sistema como um todo. Em alguns casos, a funcionalidade não desejada pode mesmo causar sérias falhas no sistema. O Quadro 19.1 descreve brevemente uma situação em que a funcionalidade desnecessária em um sistema reusado causou uma falha catastrófica de software.

O problema no lançador do Ariane 5 surgiu porque as suposições feitas sobre o software para o Ariane 4 eram inválidas para o Ariane 5. Isso é um problema comum com componentes reusáveis. Eles são originalmente implementados para um ambiente de aplicação e, naturalmente, para suposições próprias desse ambiente. Essas suposições raramente são documentadas e, assim, quando o componente é reusado, é impossível derivar testes para verificar se as suposições ainda são válidas.

19.3 Composição de componentes

A composição de componentes é o processo de montagem de componentes para criar um sistema. Se considerarmos uma situação em que componentes reusáveis estejam disponíveis, a maioria dos sistemas será construída por meio da composição desses componentes reusáveis uns com outros, com componentes especialmente escritos e com a infra-

Quadro 19.1

Falha de validação de componente.

A falha do lançador Ariane 5

Durante o desenvolvimento do lançador espacial Ariane 5, os projetistas decidiram reusar o software de referência inercial que tinha sido executado com sucesso no lançador Ariane 4. O software de referência inercial mantém a estabilidade do foguete. Eles decidiram reusá-lo sem mudanças (como você faria com componentes), embora incluísse funcionalidades além das necessárias ao Ariane 5.

No primeiro lançamento do Ariane 5, o software de navegação inercial falhou depois de 37 segundos e o foguete não podia ser controlado. Controladores em terra instruíram o lançador a se autodestruir e a carga do foguete foi destruída. Uma investigação posterior descobriu que a causa do problema foi uma exceção não tratada quando a conversão de um número em ponto fixo para um inteiro resultou em um overflow numérico. Isso fez com que o sistema de run-time desligasse o sistema de referência inercial e a estabilidade do lançador não pôde ser mantida. O defeito nunca ocorreu no Ariane 4 porque este tinha máquinas menos potentes e o valor convertido não era grande o suficiente para causar o overflow de conversão.

O defeito ocorreu em um código não necessário ao Ariane 5. Os testes de validação para o software reusado eram baseados nos requisitos do Ariane 5. Como não havia requisitos para a função que falhou, nenhum teste foi desenvolvido. Conseqüentemente, o problema com o software nunca foi descoberto durante os testes de lançamento simulado.

estrutura de apoio de componente fornecido pelo framework de modelo. Como expliquei na Seção 19.1, essa infra-estrutura fornece recursos para apoiar a comunicação de componente e serviços horizontais, como serviços de interface com o usuário, gerenciamento de transações, concorrência e proteção. As maneiras pelas quais os componentes são integrados com essa infra-estrutura são documentadas para cada modelo de componente e não serão explicadas nesta seção.

A composição não é uma operação simples; há vários tipos (Figura 19.7):

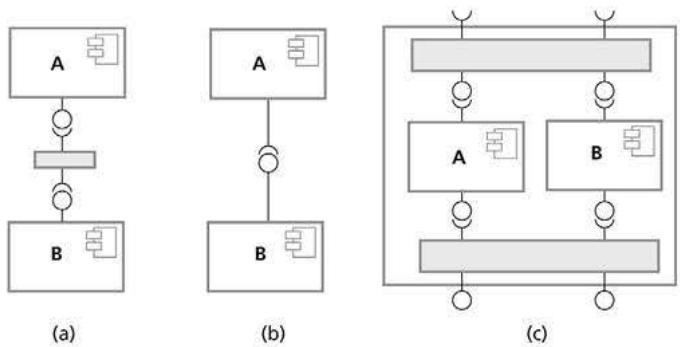
1. *Composição seqüencial.* Ocorre quando, no componente composto, os componentes constituintes são executados em seqüência. Ela corresponde à situação (a) na Figura 19.7, em que as interfaces *provides* de cada componente são compostas. Algum tipo de código extra é necessário para fazer a ligação entre os componentes.
2. *Composição hierárquica.* Ocorre quando um componente chama diretamente os serviços fornecidos por outro componente. Ela corresponde à situação em que a interface *provides* de um componente é composta com a interface *requires* de outro componente. Essa é a situação (b) na Figura 19.7.
3. *Composição aditiva.* Ocorre quando as interfaces de dois ou mais componentes são colocadas juntas (adicionadas) para criar um novo componente. As interfaces do componente composto são criadas com a junção de todas as interfaces dos componentes constituintes e a remoção das operações duplicadas, se necessário. Isso corresponde à situação (c) na Figura 19.7.

Você pode usar todas as formas de composição de componente na criação de um sistema. Em todos os casos, você pode ter de escrever 'código colante' para ligar os componentes. Por exemplo, para a composição seqüencial, a saída do componente A tipicamente torna-se a entrada do componente B. Você necessita de declarações intermediárias que chamem o componente A, coletem o resultado e, então, chamem o componente B com esse resultado como um parâmetro.

Ao escrever componentes especialmente para composição, você projeta as interfaces desses componentes de maneira que sejam compatíveis. Você pode, portanto, compor esses componentes em uma única unidade. Contudo, quando os componentes

Figura 19.7

Tipos de composição de componentes.



são desenvolvidos independentemente para reuso, você muitas vezes estará diante de incompatibilidades devido às interfaces dos componentes que você deseja compor não serem as mesmas. Três tipos de incompatibilidade podem ocorrer:

1. *Incompatibilidade de parâmetro*. As operações em cada lado da interface têm o mesmo nome, mas seus tipos de parâmetro ou o número de parâmetros são diferentes.
2. *Incompatibilidade de operação*. Os nomes de operações nas interfaces *provides* e *requires* são diferentes.
3. *Inconclusão de operação*. A interface *provides* de um componente é um subconjunto da interface *requires* de outro componente ou vice-versa.

Em todos os casos, o problema de incompatibilidade é resolvido escrevendo-se um componente adaptador que reconcilia as interfaces de dois componentes que estão sendo reusados. Quando você conhece as interfaces dos componentes que deseja usar, é possível escrever um componente adaptador que converte uma interface em outra. A forma precisa do adaptador depende do tipo de composição. Algumas vezes, como no próximo exemplo, o adaptador simplesmente toma o resultado de um componente e o converte em um formulário que pode ser usado como entrada para outro. Em outros casos, o adaptador pode ser chamado pelo componente A e ele mesmo chamar o componente B. Esta última situação surge se A e B forem compatíveis embora o número de parâmetros em suas interfaces seja diferente.

Para ilustrar os adaptadores, considere os componentes mostrados na Figura 19.8. Eles podem ser partes de um sistema usado pelos serviços de emergência. Quando o operador do serviço recebe uma chamada, o número do telefone é a entrada para o componente **addressFinder** para localizar o endereço. Então, usando o componente **mapper**, é impresso um mapa a ser enviado para o veículo despachado para a atender à emergência. De fato, os componentes têm mais interfaces complexas que as mostradas aqui, mas a versão simplificada ilustra o conceito de um adaptador.

O primeiro componente, **addressFinder**, encontra o endereço que corresponde ao número de telefone. Ele pode também retornar o dono da propriedade associada ao número de telefone e o tipo da propriedade. O componente **mapper** obtém um código postal (nos Estados Unidos, um padrão de código ZIP com quatro dígitos adicionais identificando a localização da propriedade) e mostra ou imprime o mapa da área ao redor do código em uma escala de 1:10.000.

Esses componentes são, em princípio, compostos, pois a localização da propriedade inclui o código postal ou ZIP. Contudo, você precisa escrever um componente adaptador chamado **postCodeStripper** que obtém os dados de localização do **addressFinder** e desmonta o código postal. Esse código é, depois, usado como uma entrada para o **mapper**, e o mapa da rua é mostrado em uma escala de 1:10.000. Os códigos seguintes ilustram a seqüência necessária de chamadas para implementação:

```
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000) ;
```

Outro caso no qual um componente adaptador pode ser usado é quando um componente deseja fazer uso de outro, mas há uma incompatibilidade entre as interfaces *provides* e *requires* desses componentes. Ilustrei isso na Figura 19.9, em que o componente coletor de dados é conectado a um componente sensor usando um adaptador. Ele reconcilia as interfaces *requires* do componente de coleta de dados com as interfaces *provides* do componente sensor. O componente de coleta de dados foi projetado com um mecanismo genérico *requires* que não se baseia em uma interface específica de sensor. Antecipei que seria necessário um adaptador para conectar o coletor de dados a uma interface específica de sensor.

A explicação sobre a composição de componentes considera que você pode preferir a documentação do componente se as interfaces forem compatíveis. Naturalmente, a definição da interface inclui o nome da operação e tipos de parâmetros,

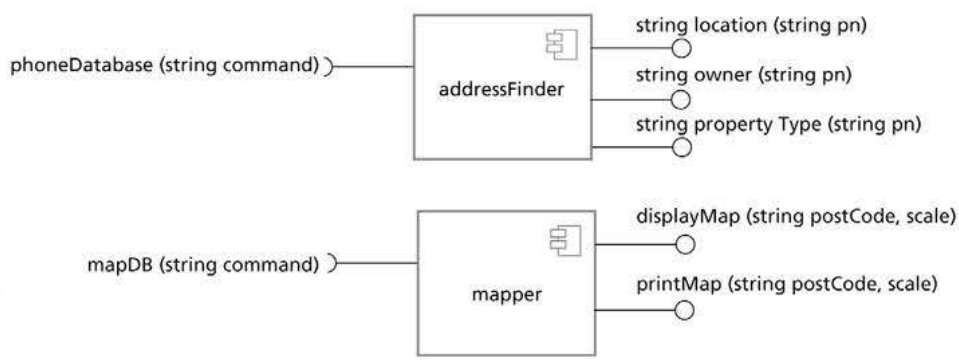
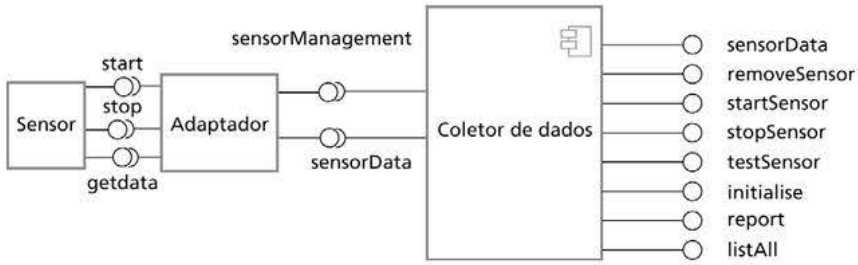


Figura 19.8
Componentes incompatíveis.

Figura 19.9

Adaptador para o componente coletor de dados.



portanto é possível fazer alguma avaliação da compatibilidade a partir disso. Contudo, você depende da documentação do componente para decidir se as interfaces são semanticamente compatíveis.

Por exemplo, considere a composição mostrada na Figura 19.10. Esses componentes são usados para implementar um sistema que baixa imagens de uma câmera digital e as armazena em um acervo de fotografias. O usuário do sistema pode fornecer informações adicionais para descrever e catalogar a fotografia. Para evitar confusão, não mostrei todos os métodos de interface aqui; simplesmente mostro os métodos necessários para ilustrar o problema da documentação de componentes. Os métodos da interface de **Photo Library** são:

public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid) ;

Considere que a documentação para o método **addItem** na **Photo Library** seja:

- Esse método adiciona uma fotografia ao acervo e associa o identificador de fotografia e o descritor de catálogo à fotografia.
- Essa descrição parece ser abrangente, mas considere as seguintes questões:
- O que acontece se o identificador da fotografia já estiver associado a uma fotografia na biblioteca?
- O descritor de fotografia está associado à entrada de catálogo, assim como a fotografia?
- Isto é, se eu excluir a fotografia, também excluirei as informações de catálogo?

Não há informações suficientes na descrição informal de **addItem** para responder a essas perguntas. Naturalmente, é possível adicionar mais informações à descrição do método em linguagem natural, mas, geralmente, a melhor maneira de resolver ambigüidades é usar uma linguagem formal para descrever a interface. No Capítulo 10, sugeri que a descrição de interface é uma área na qual especificações formais são mais úteis. A especificação mostrada na Figura 19.11 faz parte da descrição da interface de **Photo Library** que adiciona informações à descrição informal.

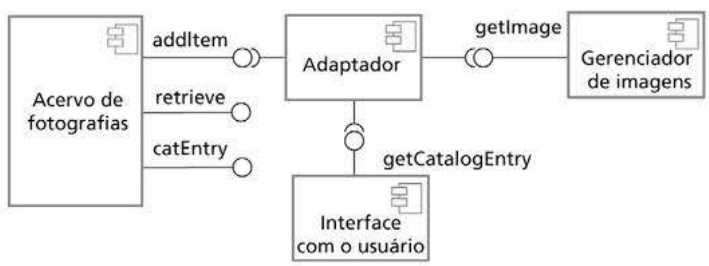
A especificação na Figura 19.11 usa pré e pós-condições; usei uma notação baseada na linguagem de restrição de objetos (OCL — Object Constraint Language) que é parte da UML (Warner e Kleppe, 1998). A OCL é estruturada para descrever restrições em modelos UML de objetos; ela permite expressar predicados que devem sempre ser verdadeiros, que devem ser verdadeiros antes da execução do método e que devem ser verdadeiros depois que o método foi executado. Estas são invariantes, pré-condições e pós-condições. Para acessar o valor de uma variável antes de uma operação, você deve adicionar **@pre** depois de seu nome. Portanto:

age = age@pre + 1

significa que o valor de age depois de uma operação é 1 a mais que antes da operação.

Figura 19.10

Composição do acervo de fotografias.



Abordagens baseadas em OCL estão sendo cada vez mais usadas para adicionar informações de semântica aos modelos UML. A abordagem geral foi derivada da abordagem de projeto por contrato (*Design by Contract*) de Meyer (Meyer, 1992), na qual as interfaces e as obrigações dos objetos que se comunicam são formalmente especificadas e reforçadas pelo sistema de run-time. Meyer sugere que o uso de projeto por contrato é essencial no desenvolvimento de componentes confiáveis (Meyer, 2003).

A Figura 19.11 apresenta uma especificação para os métodos **addItem** e **delete** em **Photo Library**. O método que está sendo especificado é indicado pelo contexto de palavras-chave e as pré e pós-condições pelas palavras-chave **pré** e **pós**. As precondições para **addItem** definem que:

- Não deve haver uma fotografia na biblioteca com o mesmo identificador quando a fotografia for inserida.
- A biblioteca deve existir — considere que a criação de uma biblioteca adiciona um único item a ela de modo que o tamanho de uma biblioteca é sempre maior do que zero.

As pós-condições para **addItem** definem que:

- O tamanho da biblioteca aumentou em 1 (portanto, somente uma única entrada foi feita).
- Se você recupera usando o mesmo identificador, então você obtém a fotografia que adicionou.
- Se você visualizar o catálogo usando esse identificador, obterá a entrada do catálogo que fez.

A especificação de **delete** fornece informações adicionais. A precondição define que, para excluir um item, ele deve estar na biblioteca e, depois da exclusão, a foto não pode mais ser recuperada e o tamanho da biblioteca será reduzido em 1. Contudo, **delete** não exclui a entrada do catálogo — você poderá ainda recuperá-la depois que a foto tiver sido excluída. A razão para isso é que você pode desejar manter no catálogo as informações de por que a foto foi excluída, sua nova localização etc.

Ao criar um sistema por composição de componentes, você pode descobrir que há conflitos potenciais entre requisitos funcionais e não funcionais, a necessidade de entregar um sistema tão rápido quanto possível e a necessidade de criar um sistema que pode se adaptar à mudança de requisitos. As decisões que envolvem assumir compromissos são:

1. Qual composição de componente é mais eficiente na entrega de requisitos funcionais para o sistema?
2. Qual composição de componentes permitirá adaptações a futuras mudanças dos requisitos?
3. Quais serão as propriedades do sistema composto? Essas propriedades emergentes são propriedades como desempenho e confiança. Você somente poderá avaliar essas propriedades após o sistema completo ter sido implementado.

Infelizmente, há muitas situações em que as soluções para os problemas de composição são mutuamente conflitantes. Por exemplo, considere uma situação como a ilustrada na Figura 19.12, na qual um sistema pode ser criado por meio de duas composições alternativas. Trata-se de um sistema de coleta de dados e geração de relatórios em que os dados são coletados de fontes diferentes, armazenados em um banco de dados, e um relatório diferente que resume esses dados é gerado.

As vantagens da composição (a) são que os gerenciamentos de geração de relatórios e de dados são separados, portanto há mais flexibilidade para mudanças futuras. O sistema de gerenciamento de dados pode ser substituído e, se o componente

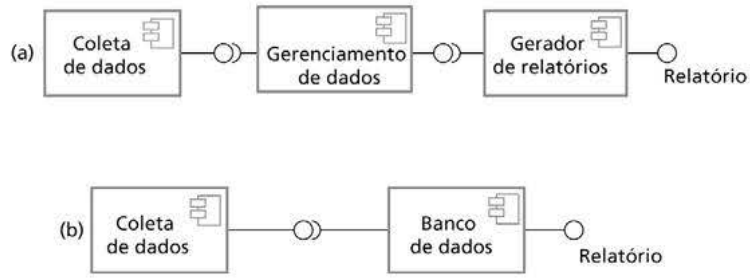
Figura 19.11

Descrição formal da interface do acervo de fotografias.

```
— A palavra-chave de contexto identifica o componente ao qual as condições se aplicam
contexto addItem

— As precondições especificam o que deve ser verdadeiro antes da execução de addItem
pré:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

— As pós-condições especificam o que é verdadeiro após a execução
pós:    libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

contexto delete
pré:    PhotoLibrary.retrieve(pid) ≠ null ;

pós:    PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre[em]1
```

Figura 19.12

Componentes de coleta de dados e geração de relatórios.



atual de geração de relatórios não puder produzir os relatórios necessários, esse componente também poderá ser substituído. Na composição (b), um componente de banco de dados com recursos built-in de geração de relatórios (por exemplo, Microsoft Access) é usado. As vantagens da composição (b) são que há poucos componentes; sendo assim, esse será provavelmente o mais rápido, pois não há overheads de comunicação de componentes. Além disso, as regras de integridade de dados que se aplicam ao banco de dados também se aplicarão aos relatórios. Esses relatórios não serão capazes de combinar dados de maneira incorreta. Na composição (a), não há tais restrições, portanto, os erros nos relatórios são mais prováveis.

Em geral, um bom princípio de composição é o da separação de assuntos. Isto é, você deve tentar projetar seu sistema de maneira que cada componente tenha um papel claro definido e que, idealmente, esses papéis não se sobreponham. Contudo, pode ser mais barato comprar um componente multifuncional do que duas ou três partes separadas. Além disso, pode haver perdas de confiança e desempenho quando múltiplos componentes são usados.

PONTOS-CHAVE

- A engenharia de software baseada em componentes é uma abordagem baseada em reuso para definição, implementação e composição de componentes independentes não firmemente acoplados nos sistemas.
- Um componente é uma unidade de software cuja funcionalidade e dependências são completamente definidas por um conjunto de interfaces públicas. Os componentes podem ser combinados com outros componentes sem referência à sua implementação e podem ser implantados como uma unidade executável.
- Um modelo de componente define um conjunto de padrões para componentes, incluindo padrões de interface, de uso e de implantação. A implementação do modelo de componente fornece um conjunto de serviços horizontais que podem ser usados por todos os componentes.
- Durante o processo CBSE, você precisa intercalar os processos de engenharia de requisitos e projeto do sistema. Você precisa estabelecer um compromisso entre os requisitos desejáveis e os serviços disponíveis nos componentes atuais reusáveis.
- A composição de componentes é o processo de junção de componentes para criar um sistema. Os tipos de composição incluem composições seqüencial, hierárquica e aditiva.
- Ao compor componentes reusáveis que não foram escritos para sua aplicação, normalmente é necessário escrever adaptadores ou 'código colante' para reconciliar as diferentes interfaces de componente.
- Ao escolher composições, você precisa considerar a funcionalidade necessária do sistema, os requisitos não funcionais e a facilidade com que um componente pode ser substituído por outro quando o sistema é alterado.

LEITURAS SUGERIDAS

Component-based software engineering: putting the pieces together. Este livro é uma coleção de artigos de vários autores sobre diferentes aspectos do CBSE. Como em todas as coleções, o texto é um pouco disperso, mas oferece uma abrangência maior de tópicos gerais da engenharia de software com componentes do que o livro de Szyperski. (G. T. Heineman e W. T. Councill, 2001, Addison-Wesley.)

Component software: beyond object-oriented programming, 2nd ed. Esta edição atualizada do primeiro livro sobre CBSE abrange tópicos técnicos e não técnicos em CBSE. Ela inclui mais detalhes sobre tecnologias específicas do que o livro de Heineman e Councill e uma explicação minuciosa sobre questões de mercado. (C. Szyperski, 2002, Addison-Wesley.)

"Specification, implementation and deployment of components". Uma boa introdução aos fundamentos de CBSE. O mesmo tópico do CACM inclui artigos sobre componentes e desenvolvimento baseado em componentes. (I. Crnkovic, et al., *Comm. ACM*, 45(10), out. 2002.)

EXERCÍCIOS

19.1 Por que é importante que todas as interações de componentes sejam definidas por interfaces *requires* e *provides*?

19.2 Os princípios de independência de componente significam que deve ser possível substituir um componente por um outro implementado de maneira completamente diferente. Usando um exemplo, explique como tal substituição de componente pode ter conseqüências indesejáveis e levar a uma falha de sistema.

19.3 Quais são as diferenças fundamentais entre componentes e Web services (veja o Capítulo 12)?

19.4 Por que é importante que os componentes sejam baseados em um modelo padronizado de componente?

19.5 Usando o exemplo de um componente que implementa um tipo abstrato de dados, como uma pilha ou uma lista, mostre por que geralmente é necessário ampliar e adaptar componentes para reuso.

19.6 Explique por que é muito difícil validar um componente reusável sem seu código-fonte. De que maneira uma especificação formal de componente pode simplificar os problemas de validação?

19.7 Projete um componente reusável que implemente a característica de busca do sistema LIBSYS explicada nos capítulos anteriores. Não se trata de uma simples busca de palavras-chave de páginas Web. Deve possibilitar buscar os catálogos de várias bibliotecas, conforme especificado pelo usuário.

19.8 Usando exemplos, ilustre os diferentes tipos de adaptadores necessários para apoiar as composições seqüencial, hierárquica e aditiva.

19.9 Projete as interfaces de componentes que podem ser usadas em um sistema em uma sala de controle de emergências. Você deve projetar interfaces para um componente *call-logging* que registra chamadas feitas e um componente *vehicle-discovery* que, dado um código postal e um tipo de incidente, localiza o veículo adequado a ser despachado para o incidente.

19.10 Foi sugerido que uma autoridade de certificação independente deve ser definida. Fornecedores devem enviar seus componentes a essa autoridade, que constatará se o componente é confiável. Quais são as vantagens e desvantagens de tal autoridade de certificação?

20

Desenvolvimento de sistemas críticos

Objetivos

O objetivo deste capítulo é apresentar técnicas de implementação usadas no desenvolvimento de sistemas críticos. Após ler este capítulo, você:

- entenderá como a prevenção e a tolerância a defeitos contribuem para o desenvolvimento de sistemas confiáveis;
- conhecerá as características e as atividades dos processos de software confiável;
- terá sido apresentado às técnicas de programação para prevenção de defeitos;
- compreenderá os estágios envolvidos na implementação de tolerância a defeitos e as maneiras pelas quais a diversidade e a redundância são usadas nas arquiteturas tolerantes a defeitos.

Conteúdo

20.1 Processos confiáveis
20.2 Programação confiável
20.3 Tolerância a defeitos
20.4 Arquiteturas tolerantes a defeitos

Técnicas de engenharia de software avançadas, linguagens de programação aprimoradas e melhor gerenciamento de qualidade conduziram a aprimoramentos significativos na confiabilidade da maioria dos softwares. Entretanto, sistemas críticos, como os que controlam máquinas não assistidas, sistemas médicos, comutadores de telecomunicações ou aeronaves necessitam de níveis mais altos de confiabilidade. Nesses casos, técnicas especiais de desenvolvimento podem ser usadas para assegurar que o sistema seja seguro, protegido e confiável.

Existem três abordagens complementares para desenvolver um software confiável:

1. *Prevenção de defeitos*. O processo de projeto e de implementação do sistema deve usar abordagens de desenvolvimento de software que ajudem a evitar erros de programação e, assim, minimizar o número de defeitos de um programa.
2. *Detecção de defeitos*. Os processos de verificação e de validação são projetados para descobrir e remover defeitos de um programa antes que este seja implantado para uso operacional.

3. *Tolerância a defeitos*. O sistema é projetado de forma que os defeitos ou o comportamento inesperado do sistema, durante a execução, sejam detectados e gerenciados de modo que a falha do sistema não ocorra.

Este capítulo enfoca os processos e as técnicas que apóiam a prevenção e a tolerância de defeitos. A detecção dos defeitos é tópico principal pela sua importância e é abordada na Parte 5. Explico técnicas estáticas para detecção de defeitos no Capítulo 22, testes de programa no Capítulo 23 e técnicas de verificação e validação específicas para os sistemas críticos no Capítulo 24.

Fundamentais para a confiabilidade de qualquer sistema são as noções básicas de redundância e diversidade. Estas são estratégias cotidianas de proteção para evitar falhas. Se você está investindo na bolsa, não deve alocar todos os seus investimentos numa única empresa, pois pode perder tudo se a empresa falir (diversidade). As pessoas guardam pilhas e lâmpadas reservas em seus lares para que possam se recuperar rapidamente de falhas (redundância). Todos nós devemos fazer back-up de nossos computadores regularmente em casos de falha no disco (redundância) e, para proteger nossos lares, geralmente temos mais de um tipo de fechadura na porta (diversidade).

Sistemas críticos podem incluir componentes que replicam a funcionalidade de outros componentes (redundância) ou código adicional de verificação que não é estritamente necessário para que o sistema funcione (redundância). Portanto, os defeitos podem ser detectados antes que causem falhas, e o sistema pode ser capaz de continuar operando caso os componentes individuais falhem. Se os componentes redundantes não forem os mesmos que outros componentes (diversidade), uma falha em comum no mesmo componente replicado não resultará na falha completa do sistema.

Em sistemas em que a disponibilidade é um requisito crítico, servidores redundantes são normalmente disponibilizados. Eles entram em operação automaticamente caso um servidor designado falhe. Algumas vezes, para assegurar que ataques ao sistema não possam explorar uma vulnerabilidade comum, os servidores podem ser de tipos diferentes e executar diferentes sistemas operacionais. O uso de sistemas operacionais diferentes é um exemplo de diversidade e de redundância de software. Em outros casos, como explico mais adiante, a diversidade e a redundância podem ser incorporadas ao software por meio da inclusão de componentes de software redundantes deliberadamente programados para usar técnicas diferentes.

A diversidade e a redundância também são usadas para obter processos confiáveis. Bem como testar um programa, você também pode usar inspeções de programa e análise estática como técnicas de descoberta de defeitos. Essas técnicas de validação são complementares: uma pode encontrar defeitos ignorados pela outra. Além disso, a mesma atividade de processo (por exemplo, uma inspeção de programa) pode ser efetuada por vários membros da equipe — as pessoas cuidam de suas tarefas de maneiras diferentes, dependendo de sua personalidade, experiência e educação, portanto esse tipo de redundância fornece uma perspectiva diversificada do sistema.

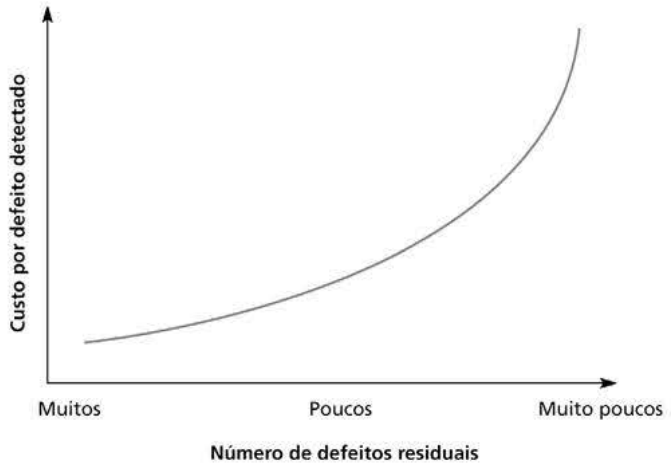
Infelizmente, incluir diversidade e redundância aos sistemas torna-os mais complexos e, dessa maneira, mais difíceis de compreender. Portanto, é mais provável que os programadores venham a cometer erros e menos provável que pessoas, ao verificar os programas, encontrem erros. Conseqüentemente, algumas pessoas consideram melhor evitar a redundância e a diversidade em um software para que o desenvolvimento do sistema seja tão simples quanto possível, e ter procedimentos de verificação e validação extremamente rigorosos (Parnas, et al., 1990). Ambas as abordagens são utilizadas em sistemas de segurança comerciais críticos. O sistema de controle de vôo do Airbus 340 é redundante e tem diversidade (Storey, 1996), enquanto o sistema de controle de vôo do Boeing 777 baseia-se em uma única versão do software.

Um dos objetivos da pesquisa em engenharia de software tem sido desenvolver ferramentas, técnicas e métodos que conduzam a uma produção de software livre de defeitos. Um software livre de defeitos é o que atende exatamente a sua especificação. Entretanto, isso não significa que o software nunca irá falhar. Podem ocorrer erros na especificação que serão refletidos no software ou os usuários podem não compreender ou usar inapropriadamente o sistema de software. No entanto, a eliminação de defeitos do software tem certamente impacto enorme no número de falhas do sistema.

Para sistemas de pequeno e médio portes, as técnicas de engenharia de software provavelmente tornam possível desenvolver software livre de defeitos. Para atingir esse objetivo, você precisa usar uma gama de técnicas de engenharia de software:

1. *Processos de software confiáveis*. O uso de um processo de software confiável (explicado na Seção 20.1) com atividades de validação e verificação apropriadas é essencial caso o número de defeitos em um programa deva ser minimizado e aqueles ignorados devam ser detectados.
2. *Gerenciamento de qualidade*. A organização que desenvolve o sistema deve ter uma cultura na qual a qualidade conduz o processo de software. A cultura deve estimular os programadores a criar programas livres de defeitos. Padrões de desenvolvimento e projeto devem ser estabelecidos e procedimentos devem estar disponíveis para verificar se os padrões foram seguidos.
3. *Especificação formal*. Deve haver uma especificação precisa (de preferência formal) de sistema que defina o sistema a ser implementado. Muitos defeitos de projeto e programação resultam de uma interpretação errada de uma especificação ambígua ou mal descrita.

Figura 20.1
Custos crescentes de remoção de defeitos residuais.



4. *Verificação estática*. Técnicas de verificação estática, como o uso de analisadores estáticos, podem encontrar características anômalas de programação que podem ser defeitos. A verificação formal, baseada na especificação do sistema, também pode ser usada.
5. *Tipagem forte*. Uma linguagem de programação com tipos fortes de dados, como Java ou ADA, deve ser usada para o desenvolvimento. Se a linguagem tiver tipagem forte, o compilador poderá detectar muitos defeitos de programação antes que sejam incluídos no programa a ser entregue.
6. *Programação segura*. Algumas construções da linguagem de programação são mais complexas e propensas a erros do que outras e será mais provável cometer erros caso sejam usadas. A programação segura significa evitar, ou ao menos minimizar, o uso dessas construções.
7. *Informações protegidas*. Deve ser usada uma abordagem para projeto e implementação de software baseada em ocultamento e encapsulamento de informações. Linguagens orientadas a objetos, como Java, obviamente, satisfazem essa condição. Deve ser encorajado o desenvolvimento de programas projetados para facilidade de leitura e compreensão.

Discuti várias dessas técnicas em outros capítulos deste livro. Neste capítulo, concentro-me em descrever processos de software confiável e técnicas de programação que contribuem para prevenção de defeitos.

Contudo, existem raras situações em que é economicamente prático implantar todas essas técnicas para criar software livre de defeitos. O custo de encontrar e remover defeitos remanescentes aumenta exponencialmente à medida que são descobertos e removidos do programa (Figura 20.1). À medida que o software se torna mais confiável, é necessário gastar mais e mais tempo e esforço para encontrar defeitos cada vez em menor número. Em determinado estágio, os custos desse esforço adicional tornam-se injustificáveis.

Como conseqüência, as empresas de desenvolvimento de software admitem que seu software sempre conterá alguns defeitos residuais. O nível de defeitos depende do tipo de sistema. Produtos comerciais têm nível relativamente alto de defeitos (embora estejam muito melhores que há dez anos), enquanto sistemas críticos têm, normalmente, uma densidade de defeitos muito menor.

A lógica para a aceitação de defeitos é que, se e quando o sistema falhar, é mais vantajoso pagar pelas conseqüências da falha do que descobrir e remover os defeitos antes da entrega do sistema. Entretanto, como explicado no Capítulo 3, a decisão de liberar o software com defeitos não é só econômica. A aceitação política e social da falha do sistema também deve ser considerada.

20.1 Processos confiáveis

Processos de software confiáveis são os voltados à prevenção e detecção de defeitos. Processos confiáveis são bem definidos e repetíveis, e incluem uma gama de atividades de verificação e validação. Um processo bem definido é um processo padronizado e documentado. Um processo repetível é o que não confia na interpretação e no julgamento individuais. Independentemente das pessoas envolvidas no processo, a organização pode ter certeza de que o processo será bem-sucedido. Explico a importância dos processos para atingir a qualidade em software e aprimoramento de processos no Capítulo 28. As características essenciais dos processos confiáveis são mostradas na Tabela 20.1.

Tabela 20.1 Características de processos confiáveis

Característica de processo	Descrição
Documentável	O processo deve ter um modelo de processo definido que estabeleça as atividades no processo e a documentação que deve ser produzida durante essas atividades.
Padronizado	Um conjunto abrangente de padrões de desenvolvimento de software que define como o software deve ser produzido e documentado deve estar disponível.
Auditável	O processo deve ser compreensível para pessoas externas ao processo que possam verificar se os padrões de processo estão sendo seguidos e fazer sugestões para o aprimoramento do processo.
Diversidade	O processo deve incluir atividades diferentes e redundantes de verificação e validação.
Robustez	O processo deve ser capaz de se recuperar das falhas de atividades individuais do processo.

Um processo confiável deve sempre incluir atividades de validação e verificação amplas e bem planejadas, cujo objetivo é assegurar que defeitos residuais no software sejam descobertos antes de sua entrega. Atividades de processo dirigidas à prevenção e detecção de defeitos incluem:

1. *Inspeções de requisitos*. Como expliquei no Capítulo 7, são destinadas a descobrir problemas com a especificação de sistema. Uma alta proporção de defeitos em software resulta de erros de requisitos. Se estes puderem ser descobertos e eliminados da especificação, então esse tipo de defeito será minimizado.
2. *Gerenciamento de requisitos*. O gerenciamento de requisitos, explicado no Capítulo 7, está relacionado a manter o acompanhamento das mudanças de requisitos ao longo do projeto e da implementação. Muitos defeitos em sistemas já entregues são resultantes de falhas em assegurar que uma mudança de requisitos foi realmente incluída no projeto e na implementação do sistema.
3. *Verificação de modelos*. A verificação de modelos envolve o uso de ferramentas CASE para análise automática de modelos de sistemas para assegurar a consistência interna e externa. *Consistência interna* significa que um modelo é consistente; *consistência externa* significa que modelos diferentes de um sistema (por exemplo, um modelo de estado e um modelo de objeto) são consistentes.
4. *Inspeções de projeto e de codificação*. Conforme explico no Capítulo 22, as inspeções de projeto e código freqüentemente baseiam-se em checklists de defeitos comuns e destinam-se a descobrir e remover esses defeitos antes do teste de sistema.
5. *Análise estática*. A análise estática é uma técnica automatizada de análise de programa na qual o programa é analisado detalhadamente para encontrar condições potencialmente errôneas. Explico isso no Capítulo 22.
6. *Planejamento e gerenciamento de teste*. Amplo conjunto de testes para o sistema deve ser projetado, e o processo de testes deve ser gerenciado cuidadosamente para assegurar a completa cobertura e rastreabilidade entre os testes do sistema e os requisitos e o projeto do sistema. Os testes são explicados no Capítulo 23.

Uma possível fonte de defeitos em sistemas críticos é incluir o componente ou a versão errada de um componente no sistema. Para evitar isso, você precisa realizar um gerenciamento de configuração rigoroso. O *gerenciamento de configuração* refere-se a gerenciar mudanças de software e manter o acompanhamento das versões de um sistema e de seus componentes. Abordo esse tópico no Capítulo 29.

20.2 Programação confiável

A programação confiável envolve o uso de construções e técnicas de programação que contribuem para prevenção e tolerância a defeitos. Os defeitos em programas surgem porque os programadores cometem erros. Enquanto alguns defeitos são devidos a erros de entendimento da especificação, outros surgem de programas excessivamente complexos ou do uso de construções inerentemente propensas a erros. Para atingir a confiabilidade, portanto, você deve elaborar o projeto levando em conta a simplicidade, a proteção de informações de acesso não autorizado e a minimização do uso de construções de programação potencialmente não seguras.

Técnicas de programação para tolerância a defeitos baseiam-se no fato de que há uma distinção entre defeitos e falhas. Uma *falha* é algo observável pelos usuários de um sistema, enquanto um *defeito* é uma característica interna do sistema. Se

um defeito se manifesta na execução de um programa, você pode ser capaz de tolerá-lo, detectando-o e executando uma ação de recuperação antes que isso resulte em uma falha do sistema. Nesta seção, explico o uso de construções de tratamento de exceções para criar programas mais tolerantes a defeitos e mais fáceis de compreender.

20.2.1 Informações protegidas

Um princípio de proteção dotado por organizações militares é o princípio do 'precisa saber'. Somente àqueles indivíduos que precisam saber de uma informação específica a fim de executarem suas obrigações é passada essa informação. Informações que não sejam diretamente relevantes a seu trabalho são negadas.

Em programação, você deve adotar um princípio análogo para controlar o acesso aos dados do sistema. Os componentes do programa devem somente ter a permissão de acesso aos dados de que precisam para sua implementação. Você pode proteger outros dados usando as regras de escopo da linguagem de programação para ocultar os dados de outras partes do programa. Se você ocultar as informações, elas não poderão ser corrompidas por componentes do programa que não devem usá-las. Se a interface permanecer a mesma, a representação dos dados pode ser mudada sem afetar outros componentes no sistema.

A proteção de informações é muito mais simples em Java do que em linguagens de programação mais antigas como C ou Pascal. Devido a essas linguagens não terem construções de encapsulamento como classes de objetos, detalhes da implementação de estruturas de dados não podem ser protegidos. Outras partes do programa podem acessar a estrutura diretamente, o que pode conduzir a efeitos colaterais inesperados quando são feitas mudanças.

Em geral, ao programar em uma linguagem orientada a objetos, é uma boa prática fornecer métodos que acessem e atualizem valores de atributos em vez de permitir que outros objetos acessem esses atributos diretamente. Isso significa que é possível mudar a representação do atributo sem se preocupar com como outros objetos usam o atributo. É particularmente importante que você use essa abordagem para estruturas de dados e outros atributos complexos.

A construção de definição da interface Java torna isso possível ao usar essa abordagem e ao declarar a interface para um objeto sem referência a sua implementação. Isso é mostrado na Figura 20.2. Usuários de objetos do tipo **Queue** podem colocar objetos na fila, removê-los da fila e consultar o tamanho da fila. No entanto, na classe que implementa essa interface, a implementação real da fila deve ser ocultada pela declaração de atributos e métodos privativos dessa classe de objeto. A separação de interfaces e sua implementação é uma parte essencial da engenharia de software baseada em componentes.

Um tipo de proteção de informações relacionado é ilustrado na Figura 20.3. Em situações em que um conjunto limitado de valores pode ser designado por alguma variável, esses valores devem ser declarados como constantes. Linguagens como C++ apóiam tipos enumerados, mas em Java isso deve ser implementado pela associação dessas restrições com a declaração de classe.

Por exemplo, considere um sistema de sinalização, implementado em Java, com luzes vermelha, amarela e verde. Um tipo **Signal** deve ser definido para incluir declarações de constantes que reflitam essas cores. Portanto, é possível se referir a **Signal.red**, **Signal.green** e assim por diante. Isso evita atribuições acidentais de valores incorretos a variáveis do tipo **Signal**. Você está, portanto protegendo variáveis do tipo **Signal** de atribuições incorretas e, ao mesmo tempo, ocultando a representação do vermelho, amarelo e verde. Você pode mudar esses valores constantes mais tarde sem ter de fazer outras mudanças no programa.

Figura 20.2

Especificação Queue que usa uma declaração de interface em Java.

```
interface Queue {

    public void put (Object o) ;
    public void remove (Object o) ;
    public int size () ;

} //Fila
```

Figura 20.3

Declaração Signal em Java que oculta a representação de tipo.

```
class Signal {

    static public final int red = 1 ;
    static public final int amber = 2 ;
    static public final int green = 3 ;

    public int sigState ;
}
```

20.2.2 Programação segura

Defeitos em programas e, portanto, muitas falhas de programa, são geralmente uma conseqüência de erro humano. Programadores cometem erros porque perdem o controle de todas as relações entre as variáveis de estado. Eles escrevem comandos de programas que resultam em comportamento inesperado e em mudanças de estado do sistema. As pessoas sempre cometerão erros, mas tornou-se claro nos anos 1960 que algumas abordagens de programação eram mais propensas a erros do que outras.

Dijkstra (Dijkstra, 1968) reconheceu que o comando *goto* ou o desvio incondicional era uma construção de programação inerentemente propensa a erros. Isso tornou difícil localizar mudanças de estados. Essa observação conduziu ao desenvolvimento de programação estruturada. Programação estruturada é uma programação sem comandos *goto*, usando somente loops e comandos *if* como construções de controle e um projeto que usa uma abordagem *top-down*. A programação estruturada foi um marco importante no desenvolvimento da engenharia de software pois foi o primeiro passo para sair da abordagem indisciplinada de desenvolvimento de software.

Outras construções de linguagem de programação e técnicas de programação também são inerentemente propensas a erros. Se você evitá-las ou, ao menos, usá-las o menos possível, será menos provável que sejam introduzidos defeitos em programas. Construções potencialmente propensas a erros incluem:

1. *Números em ponto flutuante*. Os números em ponto flutuante são inerentemente imprecisos. Este é um problema específico pois, quando são comparados, a imprecisão de representação pode conduzir a comparações inválidas. Por exemplo, 3,00000000 pode, algumas vezes, ser representado como 2,99999999 e algumas vezes como 3,00000001. Uma comparação mostraria esses números como desiguais. Números em ponto fixo, em que um número é representado para um dado número de casas decimais, são mais seguros, pois são possíveis comparações exatas.
2. *Ponteiros*. Ponteiros são construções de baixo nível que contêm endereços que se referem diretamente a áreas da memória da máquina. Erros em seu uso podem ser devastadores, pois permitem o uso de 'alias' (explicados mais adiante nesta lista) e porque tornam a verificação de limites de vetores e outras estruturas difíceis de serem implementadas.
3. *Alocação de memória dinâmica*. A memória do programa pode ser alocada durante o tempo de execução em vez de em tempo de compilação. O perigo disso é que a memória pode não ser liberada; assim, eventualmente, o sistema pode ficar sem memória disponível. Esse pode ser um defeito muito difícil de ser detectado, pois o sistema pode ser executado com sucesso por um longo tempo antes da ocorrência do problema.
4. *Paralelismo*. O paralelismo é perigoso devido às dificuldades em se prever os efeitos sutis das interações síncronas entre os processos paralelos. Problemas de timing não podem normalmente ser detectados por inspeção de programa, e a combinação peculiar de circunstâncias que causam o problema de timing pode não ocorrer durante o teste do sistema. O paralelismo pode ser inevitável, mas seu uso deve ser cuidadosamente controlado para minimizar dependências entre processos. Recursos de linguagem de programação como os threads Java ajudam a gerenciar o paralelismo de modo que alguns erros de programação possam ser evitados.
5. *Recursão*. A recursão ocorre quando um procedimento ou método chama a si mesmo ou um outro procedimento que, então, chama o procedimento original que o chamou. Seu uso pode resultar em programas concisos, mas pode ser difícil seguir a lógica de programas recursivos. Erros de programação são, portanto, mais difíceis de serem detectados. Os erros de recursão podem resultar em alocação de toda a memória do sistema à medida que variáveis de pilhas temporárias são criadas.
6. *Interrupções*. Elas são um meio de forçar a transferência do controle para uma seção de código diferente da execução atual. O perigo disso é óbvio: a interrupção pode causar o término de uma operação crítica.
7. *Herança*. O problema com a herança em programação orientada a objetos é que o código associado a um objeto não está completamente em um único lugar. Isso torna mais difícil compreender o comportamento do objeto. Dessa maneira, é mais provável que defeitos de programação sejam perdidos. Além disso, a herança, quando combinada com a ligação dinâmica, pode causar problemas de timing durante a execução. Em tempos diferentes, instâncias diferentes de um método específico podem ser chamadas e quantidades diferentes de tempo serão gastas na procura pela instância correta do método.
8. *Construções do tipo 'alias'*. Ocorre quando mais do que um único nome é usado para se referir a uma mesma entidade em um programa. É fácil para os leitores de programa perder comandos que mudam a entidade quando há vários nomes a considerar.
9. *Vetores não limitados*. Em linguagens como C, os vetores são simplesmente caminhos de acesso à memória, e você pode criar atribuições além do fim de um vetor. O sistema de run-time não verifica se os registros realmente se

referem aos elementos contidos no vetor. O overflow de buffer, no qual um invasor constrói deliberadamente um programa que escreve na memória além do fim de um buffer implementado como um vetor, é uma vulnerabilidade de proteção conhecida.

10. *Processamento de entradas default*. Alguns sistemas fornecem um default para processamento de entradas sem considerar a entrada apresentada ao sistema. Isso constitui um defeito de proteção que um invasor pode explorar pela apresentação de entradas inesperadas ao programa que não são rejeitadas pelo sistema.

Alguns padrões para desenvolvimento de sistemas críticos de segurança proíbem completamente o uso dessas construções. Contudo, essa postura extrema normalmente não é prática. Todas essas construções e técnicas são úteis, mas devem ser utilizadas com cuidado. Sempre que possível, seus efeitos potencialmente perigosos devem ser controlados pelo uso destes dentro de tipos abstratos de dados ou de objetos. Elas atuam como firewalls limitando os danos causados caso ocorram defeitos.

Os projetistas do Java reconheceram alguns dos problemas de construções propensas a erros. A linguagem não inclui comandos *goto*; ela tem um built-in de coleta de lixo de modo que não há necessidade de alocação de memória dinâmica, e ela não apóia ponteiros ou vetores não limitados. No entanto, a representação numérica do Java permite que o overflow não seja detectado pelo sistema de run-time, e as falhas devidas aos defeitos de números em ponto flutuante ainda são possíveis de acontecer.

20.2.3 Tratamento de exceções

Durante a execução do programa, inevitavelmente ocorrem erros e eventos inesperados. Eles podem surgir devido a um defeito de programa ou podem ser resultado de circunstâncias externas imprevisíveis. Um erro ou um evento inesperado que ocorra durante a execução de um programa é denominado *exceção*. Exceções podem ser causadas por condições do hardware ou do software. Exemplos de exceções podem ser uma falha de energia do sistema, uma tentativa de acessar dados não existentes e overflow e underflow numéricos.

Quando ocorre uma exceção, esta deve ser gerenciada pelo sistema. Isso pode ser feito dentro do próprio programa ou pode envolver a transferência de controle para um mecanismo de tratamento de exceções do sistema. Tipicamente, o mecanismo de gerenciamento de exceções do sistema simplesmente relata o erro e pára a execução. Portanto, para assegurar que as exceções não causem falhas no sistema, você deve definir um tratador de exceções para todas as exceções possíveis que podem surgir e assegurar que todas as exceções sejam manipuladas explicitamente.

Em linguagens de programação como C, comandos *if* devem ser utilizados para detectar a exceção e para transferir o controle para o código de tratamento de exceções. Isso significa que você precisa verificar explicitamente as exceções onde quer que possam ocorrer no programa. Entretanto, tal fato adiciona complexidade e, assim, aumentam-se as chances de se cometer erros e de que a exceção não seja corretamente tratada.

Figura 20.4

Exceções para tratar uma falha na bomba de insulina.

```
class SensorFailureException extends Exception {

    SensorFailureException (String msg) {
        super (msg) ;
        Alarm.activate (msg) ;
    }
} // SensorFailureException

class Sensor {

    int readVal () throws SensorFailureException {

    try {
        int theValue = DeviceIO.readInteger () ;
        if (theValue < 0)
            throw new SensorFailureException ("Sensor failure") ;
        return theValue ;
    }
    catch (deviceIOException e)
        {
            throw new SensorFailureException (" Sensor read error ") ;
        }
    } // readVal
} // Sensor
```

Algumas linguagens de programação, como Java, C++ e Ada, incluem construções que apóiam o tratamento de exceções de modo que não haja necessidade de comandos condicionais extras para se verificá-las. Em vez disso, a linguagem de programação inclui um tipo built-in (freqüentemente denominado **Exception**), e exceções diferentes podem ser declaradas como se fossem desse tipo. Quando uma situação de exceção ocorre, ela é sinalizada e o sistema em run-time da linguagem transfere o controle para um tratador de exceções. Ou seja, uma seção do código que estabelece nomes de exceções e as ações apropriadas para tratar cada exceção.

Em Java, novos tipos de exceção podem ser declarados por meio da extensão da classe built-in **Exception**. Exceções são sinalizadas em Java pelo uso de um comando **throw**. O tratador de uma exceção é indicado pela palavra-chave **catch**, seguida por um bloco de códigos que podem lidar com a exceção.

A Figura 20.4 ilustra o uso de exceções em Java. Este exemplo, parte do software para bomba de insulina apresentado no Capítulo 3, é um controlador de sensor que lê o valor de glicose no sangue de um sensor. A primeira declaração na Figura 20.4 mostra como exceções em Java são declaradas. A classe de objeto built-in denominada **Exception** é estendida, e o método construtor define o código a ser implementado quando a exceção é lançada. Nesse caso, um alarme é ativado.

A classe **Sensor** fornece um método único, denominado **readVal**, que inclui um comando **throw** em sua declaração. Isso significa que uma **SensorFailureException** pode ser lançada de dentro do método, mas se espera que o método chamador forneça um tratador para **SensorFailureException**. Normalmente, é melhor que as exceções sejam tratadas pelo método chamador, pois esse método sabe o que deve ser feito com o resultado do método chamado. No entanto, como mostro mais adiante, existem algumas situações em que as exceções são tratadas localmente para assegurar que o resultado de uma chamada de método seja sempre válido.

A palavra-chave **try** indica que uma exceção pode ser lançada no bloco de código seguinte. A exceção **SensorFailureException** será lançada caso um valor menor do que zero for retornado quando o sensor for verificado. **DeviceIO.readInteger** pode lançar uma exceção denominada **deviceIOException**, assim um tratador para isso deve ser incluído seguindo a palavra-chave **catch**. Nesse caso, o controlador simplesmente lança uma exceção de falha de sensor para indicar que o objeto chamador deve tratar a exceção.

Figura 20.5

Exceções em um controlador de temperatura de freezer.

```
class FreezerController {

    Sensor tempSensor = new Sensor () ;
    Dial tempDial = new Dial () ;
    float freezerTemp = tempSensor.readVal () ;
    final float dangerTemp = (float) –18.0 ;
    final long coolingTime = (long) 200000.0 ;

    public void run ( ) throws InterruptedException {
    try {
        Pump.switchIt (Pump.on) ;
        do {
            if (freezerTemp > tempDial.setting ())
                if (Pump.status == Pump.off)
                {
                    Pump.switchIt (Pump.on) ;
                    Thread.sleep (coolingTime) ;
                }
            else
                    if (Pump.status == Pump.on)
                        Pump.switchIt (Pump.off) ;

            if (freezerTemp > dangerTemp)
                throw new FreezerTooHotException () ;
        freezerTemp = tempSensor.readVal () ;

    } while (true) ;

    } // try block
    catch (FreezerTooHotException f)
    {  Alarm.activate ( ) ; }
    catch (InterruptedException e)
    {
        System.out.println ("Thread exception") ;
        throw new InterruptedException ( ) ;
    }

    } //run
} // FreezerController
```

O tratamento de exceções também pode ser usado para simplificar programas e torná-los mais fáceis de ler e compreender. Isso reduz a probabilidade de erros do programador e aumenta as chances de os inspetores de programa descobrirem quaisquer problemas existentes. Você usa o tratamento de exceções para separar o código de tratamento de erros do código que trata do processamento normal. Portanto, pode ler e compreender cada uma dessas seções de códigos isoladamente.

Isso é mostrado na Figura 20.5. Essa classe Java é uma implementação de um controlador de temperatura de um freezer de alimentos. A temperatura necessária pode ser estabelecida entre –18 e –40 graus Celsius. As comidas congeladas começam a descongelar e bactérias tornam-se ativas a temperaturas superiores a –18 graus. O sistema de controle mantém essa temperatura alternando o interruptor de uma bomba de refrigeração de acordo com o valor de um sensor de temperatura. Se a temperatura necessária não puder ser mantida, o controlador acionará um alarme.

Na implementação Java, a temperatura do freezer é descoberta perguntando-se a um objeto denominado **tempSensor**, e a temperatura necessária é descoberta por meio da inspeção de um objeto denominado **tempDial**. Um objeto bomba (**Pump**) responde a sinais para alternar seu estado. Uma vez que a bomba tenha sido ligada, o sistema vai esperar por algum tempo (chamando **Thread.sleep**) até que a temperatura caia. Se ela não cair suficientemente, uma exceção denominada **FreezerTooHotException** será lançada.

O tratador de exceções (localizado no final do código) capta essa exceção e ativa o objeto **Alarm**. Um tratador também é incluído pela exceção **InterrruptedException** built-in, que pode ser lançada pela **Thread.sleep**. Isso registra a exceção e, depois, relança-a para tratamento por meio do método principal.

20.3 Tolerância a defeitos

Um sistema tolerante a defeitos pode continuar em operação após terem ocorrido alguns defeitos do sistema. Os mecanismos de tolerância a defeitos de um sistema asseguram que esses defeitos não causem falha do sistema. A tolerância a defeitos pode ser necessária em situações em que a falha do sistema poderia causar um acidente catastrófico ou em que uma perda da operação do sistema causaria grandes prejuízos econômicos. Por exemplo, os computadores de uma aeronave devem continuar trabalhando até ela ter pousado; os computadores de um sistema de controle de tráfego aéreo devem estar continuamente disponíveis enquanto os aviões estiverem no ar.

Existem quatro aspectos de tolerância a defeitos:

1. *Detecção de defeitos*. O sistema deve detectar um defeito que pode conduzir a uma falha de sistema. Em geral, isso envolve a verificação de se o estado do sistema é consistente.
2. *Avaliação de danos*. As partes do estado do sistema afetadas pelo defeito devem ser detectadas.
3. *Recuperação de defeitos*. O sistema deve restaurar seu estado para o estado 'seguro' conhecido. Esse estado pode ser atingido pela correção do estado de dano (recuperação forward de defeito) ou pela restauração do sistema para um estado 'seguro' conhecido (recuperação backward de defeito).
4. *Reparação de defeitos*. Envolve a modificação do sistema de maneira que o defeito não ocorra. Entretanto, muitos defeitos de software manifestam-se como estados transientes. Eles são devidos a uma combinação peculiar de entradas do sistema. Nenhum reparo é necessário e o processamento normal pode ser retomado imediatamente após a recuperação do defeito.

Você pode pensar que recursos para tolerância a defeitos são desnecessários em sistemas desenvolvidos que usam técnicas que evitam a introdução de defeitos. Se não há defeitos no sistema, não parece haver nenhuma chance de falha de sistema. Entretanto, 'livre de defeitos' não significa 'livre de falhas'. Isso pode significar somente que o programa corresponde a sua especificação. A especificação pode conter erros ou omissões e pode basear-se em suposições incorretas acerca do ambiente do sistema. E, evidentemente, nunca podemos demonstrar conclusivamente que um sistema está completamente livre de defeitos. Em sistemas com requisitos de alta confiabilidade e disponibilidade, é necessário usar abordagens diversas e redundantes de prevenção, e tolerância a defeitos.

20.3.1 Detecção de defeitos e avaliação de danos

O primeiro estágio da tolerância a defeitos é detectar se um defeito (um estado errôneo do sistema) ocorreu ou ocorrerá a não ser que alguma ação seja tomada imediatamente. Para fazer isso, você precisa saber quando o valor de uma variável de estado é ilegal ou quando relacionamentos entre variáveis de estados não são mantidos. Portanto, é preciso definir restrições de estados que definem as condições que devem sempre ser mantidas para todos os estados legais. Se esses predicados forem falsos, ocorreu um defeito. Alguns exemplos de restrições de estados que se aplicam ao software

da bomba de insulina são mostrados na Figura 20.6. Não os escrevi, deliberadamente, como comandos assert do Java por razões que explicarei mais adiante.

Existem dois tipos de detecção de defeitos que você pode usar:

1. *Detecção preventiva de defeitos*. Neste caso, o mecanismo de detecção de defeitos é iniciado antes que uma mudança de estado seja reconhecida. Se um estado potencialmente errôneo for detectado, a mudança de estado não será feita.
2. *Detecção backward de defeitos*. Neste caso, o mecanismo de detecção de defeitos é iniciado após o estado do sistema ter sido mudado para verificar se ocorreu um defeito. Se for descoberto um defeito, uma exceção será sinalizada e um mecanismo de reparo será usado para se recuperar do defeito.

Você pode usar a detecção preventiva de defeitos quando as restrições de estado definidas aplicarem-se somente a variáveis individuais de estado. Por exemplo, você pode usar essa abordagem quando o valor da variável de estado tiver de cair dentro de uma faixa definida. A detecção preventiva de defeitos evita o overhead para reparos de danos, assim o estado de sistema será sempre válido — embora não necessariamente correto. Contudo, o sistema deve ter um mecanismo para a continuidade da operação na presença de algum estado incorreto caso uma falha do sistema deva ser evitada.

Em Java, o modo mais seguro de implementar uma detecção preventiva de defeitos é verificar explicitamente os defeitos e usar o mecanismo de tratamento de exceções da linguagem para sinalizar que um estado de sistema errôneo foi detectado. Isso é ilustrado na Figura 20.7. Essa é uma implementação de uma classe em que os valores das instâncias da classe são restritos aos números pares positivos. Se for feita uma tentativa de atribuir um número ímpar ou menor do que 0, uma exceção será lançada.

Figura 20.6

Restrições de estado que se aplicam à bomba de insulina.

```
// A dose de insulina a ser liberada deve ser sempre maior
// do que zero e menor do que alguma dose máxima única definida

insulin_dose >= 0 & insulin_dose <= insulin_reservoir_contents

// A quantidade total de insulina liberada em um dia deve ser menor
// do que ou igual à dose máxima diária definida

cumulative_dose <= maximum_daily_dose
```

Figura 20.7

Classe número PositiveEven em Java.

```
class PositiveEvenInteger {

    int val = 0 ;

    PositiveEvenInteger (int n) throws NumericException
    {
        if (n < 0 l n%2 = 1)
            throw new NumericException () ;
        else
            val = n ;

    }// PositiveEvenInteger

    public void assign (int n) throws NumericException
    {
        if (n < 0 l n%2 = 1)
            throw new NumericException ();
        else
            val = n ;
    } // assign

    int toInteger ()
    {
        return val ;
    } //to Integer

    boolean equals (PositiveEvenInteger n)
    {
        return (val = n.val) ;
    } // equals

} //PositiveEven
```

Em Java 1.4, foi introduzido um recurso de asserção no qual as restrições de estados podem ser definidas em um comando assert. Assim, para especificar que um número deve ser positivo e par, você escreveria:

```
assert n >= 0 & n%2 = 0: "Valor deve ser positivo e par"
```

O sistema em run-time verifica se a condição permanece e, em caso negativo, levanta um erro e causa a impressão da mensagem associada.

Entretanto, um recurso de asserção da linguagem Java foi, realmente, projetado para ajudar a descobrir inconsistências de estado durante o desenvolvimento e depuração, mais do que dar apoio à programação tolerante a defeitos. É possível ligar ou desligar a verificação de asserção e, dessa maneira você não pode confiar que as asserções estejam sempre sendo verificadas. Além disso, não é possível associar um tipo específico de exceção a cada asserção, e, portanto, não se pode identificar a falha individual de asserção. Essa foi uma decisão de projeto deliberada — os projetistas não quiseram que fosse possível executar ações de recuperação após uma falha de asserção.

A detecção preventiva de defeitos é possível quando se conhece a faixa de valores que podem ser atribuídos a uma variável de estado. Entretanto, quando um valor válido depende do valor atribuído a outros valores de estado, a detecção preventiva de defeitos pode ser impossível. Por exemplo, digamos que seu programa leia três valores, **A**, **B** e **C**, nessa ordem. A restrição de estado será:

```
A < B & B < C
```

Você não pode aplicar a detecção preventiva de defeitos na leitura do valor de **A** porque não se sabe quais serão os valores de **B** e **C**. Da mesma maneira, quando se lê **B**, você não pode verificar se ele é menor do que **C**. Portanto, você precisa usar a detecção backward de defeitos e verificar a restrição de estado após todos os valores terem sido lidos. Caso a restrição seja falsa, você poderá executar alguma ação para restaurar a consistência do sistema.

Uma maneira de implementar a detecção backward de defeitos em Java é associar uma função de verificação a um objeto. Essa função pode ser chamada após as mudanças de estado terem sido feitas para assegurar que as restrições de estado se aplicam. Você pode chamar essas funções quando necessário — o estado pode não precisar ser verificado após cada mudança feita. A seguinte interface pode ser usada para as funções de verificação:

```
interface CheckableObject {
    public boolean check () ;
}
```

Os objetos a serem verificados são instanciações de uma classe de objeto que implementa essa interface, assim cada objeto tem uma função de verificação associada. Cada classe de objeto implementa sua própria função de verificação que define as restrições específicas que se aplicam aos objetos da classe. Quando o estado como um todo for verificado, a ligação dinâmica será usada para assegurar que a função apropriada de verificação seja aplicada à classe de objeto verificada. Podemos ver um exemplo disso na Figura 20.8, na qual a função de verificação averigua se os elementos de um vetor satisfazem alguma restrição.

A detecção backward de defeitos, que usa restrições de estado que se aplicam a mais de uma variável de estado, é mostrada na Figura 20.9. Nesse exemplo, a verificação de detecção de defeitos é aplicada a elementos consecutivos de um vetor e verifica se o vetor está ordenado.

A avaliação de danos envolve a análise de estado do sistema para estimar a extensão do quanto foi corrompido. A avaliação de danos é necessária quando não se pode evitar a mudança de estado ou quando um defeito é causado por uma seqüência inválida de mudanças de estado individualmente corretas.

O papel dos procedimentos de avaliação de danos não é se recuperar de um defeito, mas avaliar quais partes do espaço de estados foram afetadas pelo defeito. O dano pode somente ser avaliado se for possível aplicar alguma 'função de validação' que verifique se o estado é consistente. Se forem encontradas inconsistências, elas serão destacadas ou sinalizadas de alguma maneira.

A Figura 20.8 mostra uma maneira de implementar a avaliação de danos em Java. A estrutura de dados denominada **RobustArray** é um conjunto de objetos do tipo **CheckableObject**. A classe que implementa o tipo **CheckableObject** deve incluir um método denominado **check**, que pode testar se o valor do objeto satisfaz alguma restrição. Esse método de verificação é associado a esse objeto em vez de ao objeto **RobustArray**, pois os detalhes da verificação dependem do uso do tipo **CheckableObject**.

O método **assessDamage** da classe **RobustArray** examina cada elemento do vetor e verifica se seu estado está correto. Se um ou mais elementos do vetor não atenderem às restrições de estado definidas na função de verificação, os elementos danificados serão gravados no vetor **checkState**. Uma exceção denominada **ArrayDamagedException** será então lançada. Um tratador de exceções que gerencia o dano deve ser incluído no método chamador. Este pode usar as informações contidas no **checkState** para decidir o que fazer.

Figura 20.8

Classe de vetor com avaliação de danos.

```
class RobustArray {

    // Verifica se todos os objetos em um vetor de objetos
    // atendem a alguma restrição definida

    boolean [] checkState ;
    CheckableObject [] theRobustArray ;

    RobustArray (CheckableObject [] theArray)
    {
        checkState = new boolean [theArray.length] ;
        theRobustArray = theArray ;
    } //RobustArray

    public void assessDamage () throws ArrayDamagedException
    {
        boolean hasBeenDamaged = false ;

        for (int i= 0; i <this.theRobustArray.length ; i ++)
        {
            if (! theRobustArray [i].check ())
                {
                    checkState [i] = true ;
                    hasBeenDamaged = true ;
                }
            else
                checkState [i] = false ;
        }
        if (hasBeenDamaged)
            throw new ArrayDamagedException () ;
    } //assessDamage

} // RobustArray
```

Figura 20.9

Procedimento de ordenação Safe com recuperação de erros backward.

```
class SafeSort {

    static void sort ( int [] intarray, int order ) throws SortError
    {
        int [] copy = new int [intarray.length];

        // copiar o vetor de entrada

        for (int i = 0; i < intarray.length ; i++)
            copy [i] = intarray [i] ;
        try {
            Sort.bubblesort (intarray, intarray.length, order) ;
            if (order = Sort.ascending)
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i] > intarray [i+1])
                        throw new SortError () ;
            else
                for (int i = 0; i <= intarray.length-2 ; i++)
                    if (intarray [i+1] > intarray [i])
                        throw new SortError () ;
        } // try block
        catch (SortError e )
        {
            for (int i = 0; i < intarray.length ; i++)
                intarray [i] = copy [i] ;
            throw new SortError ("Array not sorted") ;
        } //catch
    } // sort
} // SafeSort
```

Outras técnicas de detecção de defeitos e de avaliação de danos dependem da representação de estado do sistema e do tipo de aplicação. Essas técnicas de avaliação de danos incluem:

1. O uso de verificação de códigos e checksums em comunicações de dados e dígitos de verificação em dados numéricos

2. O uso de ligações redundantes em estruturas de dados que contêm ponteiros
3. O uso de timers do tipo watchdog em sistemas concorrentes

Verificações de códigos (Fujiwara e Pradhan, 1990) podem ser usadas quando são trocados dados e quando o checksum está associado a dados numéricos. Um checksum é um valor único calculado por meio da aplicação de alguma função matemática aos dados. O checksum é calculado pelo emissor dos dados, o qual aplica a função de checksum aos dados e anexa esse valor de função aos dados a serem transferidos. O receptor aplica a mesma função aos dados e compara o valor calculado ao checksum anexado. Como as funções são as mesmas, se esses valores diferirem, os próprios dados foram mudados. Essa técnica pode ser usada para detectar invasões de proteção assim como para deliberar e detectar corrompimento acidental de dados.

Quando são usadas estruturas de dados ligados, a representação pode ser feita de maneira redundante por meio da inclusão de referências backward. Isto é, para toda referência de A para B, existe uma referência comparável de B para A. Você pode também contar o número de elementos na estrutura. A verificação pode determinar se as referências forward e backward são consistentes (elas devem se referir uma à outra) e se o tamanho armazenado e o tamanho da estrutura calculada são os mesmos.

Quando os processos devem reagir dentro de um período específico de tempo, um timer do tipo watchdog pode ser instalado. Esse timer deve ser redefinido pelo processo em execução após sua ação estar completa. Ele é iniciado simultaneamente a um processo e mede o tempo de execução do processo. Um controlador pode interrogá-lo em intervalos regulares. Se, por alguma razão, o processo falhar em terminar, o timer não será redefinido. O controlador pode, portanto, detectar a ocorrência de um problema e tomar a ação para forçar o término do processo.

20.3.2 Recuperação e reparo de defeitos

A recuperação de defeitos é um processo de modificação do espaço de estado do sistema de tal modo que os efeitos do defeito sejam eliminados ou reduzidos. O sistema pode continuar em operação, talvez de maneira degradada. A recuperação forward envolve tentar corrigir o estado do sistema danificado e criar o estado desejado. A recuperação backward restaura o estado do sistema para um estado 'correto' conhecido.

A recuperação de erros forward somente é possível em situações em que as informações de estado incluam redundância built-in. Existem duas situações gerais (ambas explicadas na seção anterior) nas quais essa técnica de recuperação de defeitos pode ser aplicada:

1. *Quando dados codificados são corrompidos*. O uso de técnicas de codificação que adicionam redundância aos dados permite que os defeitos sejam corrigidos bem como detectados.
2. *Quando estruturas ligadas são corrompidas*. Quando os ponteiros forward e backward são incluídos na estrutura de dados, a estrutura pode ser recriada — caso ponteiros suficientes permaneçam não corrompidos. Essa técnica é freqüentemente usada para reparar o sistema de arquivos ou banco de dados.

A recuperação de erros backward é uma técnica mais simples que restaura o estado para um estado de segurança conhecido após um defeito ter sido detectado. A maioria dos sistemas de banco de dados inclui recuperação de erros backward. Quando um usuário inicia uma computação de banco de dados, uma transação é iniciada. Mudanças feitas durante essa transação não são imediatamente incorporadas ao banco de dados. O banco de dados somente é atualizado após a transação ter terminado e nenhum problema ter sido detectado. Se a transação falhar, o banco de dados não será atualizado.

As transações permitem a recuperação do defeito porque não confirmam as mudanças no banco de dados até que elas tenham sidos completadas. Entretanto, essas transações não permitem a recuperação de mudanças de estado que sejam válidas, mas incorretas. Checkpointing é uma técnica que permite recuperar essa situação. O estado do sistema é duplicado periodicamente. Quando é descoberto um problema, um estado correto pode ser restaurado com base em uma dessas cópias.

Para exemplificar como uma recuperação backward pode ser implementada usando o chekpointing, considere a classe **SafeSort** em Java, mostrada na Figura 20.9, que inclui código para detecção de defeitos e recuperação backward.

O método cria um checkpoint por meio da cópia do vetor antes da operação de ordenação. Neste exemplo, uso um bubble sort para simplificar, mas qualquer algoritmo de ordenação pode ser usado, obviamente. Se houver um erro no algoritmo de ordenação e o vetor não for ordenado apropriadamente, tal fato será detectado por verificação explícita na ordem dos elementos no vetor. Se o vetor não for ordenado apropriadamente, uma exceção **SortError** será lançada. O tratador de exceções não tenta reparar o problema, mas restaura o valor original do vetor e relança o **SortError** para indicar ao método chamador que a ordenação não foi bem-sucedida. Então, será de responsabilidade do método chamador decidir como continuar a execução.

Como sugeri anteriormente, muitos defeitos de software são transitórios, e nenhum reparo explícito é necessário para corrigir as condições que causaram esses erros. Elas desaparecem em uma execução subseqüente do sistema. Quando esse

não for o caso, será possível executar algumas ações de reparo. A ação de reparo mais comum é reiniciar o sistema, redefinindo o estado para seus valores iniciais seguros (Huang e Kintala, 1993). Algumas vezes, isso pode ser feito sem parar o sistema, caso a inicialização seja rápida e as requisições de serviço possam ser adiadas. Outras alternativas de reparo, como a reconfiguração dinâmica, normalmente serão possíveis somente quando houver provisão explícita para isso no projeto do sistema.

20.4 Arquiteturas tolerantes a defeitos

Em muitos sistemas, é possível implementar a tolerância a defeitos de software por meio da inclusão explícita de ações de verificação e recuperação no software. Isso é denominado programação defensiva. Contudo, essa abordagem não trabalha eficientemente com defeitos do sistema que surgem com base em interações entre o hardware e o software. Além disso, a má compreensão dos requisitos pode significar que tanto o código do sistema quanto a defesa associada estão incorretos.

Para a maioria dos sistemas críticos, particularmente aqueles com requisitos rigorosos de disponibilidade, pode ser necessária uma arquitetura específica projetada para apoiar a tolerância a defeitos. Exemplos de sistemas que usam essa abordagem de tolerância a defeitos são os sistemas da aviação, que devem estar em operação durante o vôo; os sistemas de telecomunicações; e os sistemas críticos de comando e controle. Pullum (Pullum, 2001) descreve diferentes tipos de arquitetura tolerante a defeitos.

Há muitos anos existe a necessidade de se construir hardware tolerante a defeitos. A técnica de hardware tolerante a defeitos mais comum é baseada na noção de redundância modular tripla (TMR — *Triple-Modular Redundancy*). A unidade de hardware é replicada três (ou algumas mais) vezes. A saída de cada unidade passa para um comparador de saída normalmente implementado como um sistema de votação. Se uma das unidades falha e não produz a mesma saída que as outras unidades, sua saída é ignorada. Um gerenciador de defeitos pode tentar reparar a unidade defeituosa automaticamente, mas, se isso for impossível, o sistema será automaticamente reconfigurado para tirar a unidade de serviço. Em seguida, o sistema continuará a funcionar com duas unidades operando (Figura 20.10).

Essa abordagem de tolerância a defeitos baseia-se no fato de que a maioria das falhas de hardware é resultante das falhas de componente em vez de ser ocasionada por defeitos de projeto. Os componentes são, portanto, propensos a falhar independentemente. Supõe-se que, quando completamente operacionais, todas as unidades de hardware operam de acordo com a especificação. Há, portanto, baixa probabilidade de falha simultânea de componentes em todas as unidades de hardware.

Naturalmente, todos os componentes poderiam ter um defeito de projeto em comum e, assim, todos produziriam a mesma resposta (errada). Usando unidades de hardware com uma especificação em comum, mas projetadas e construídas por fabricantes diferentes, reduzem-se as chances de uma falha de modo comum. Presume-se que a probabilidade de equipes diferentes cometerem o mesmo erro de projeto ou de fabricação seja pequena.

Se os requisitos de disponibilidade e confiabilidade de um sistema forem tais que você necessite usar um hardware tolerante a falhas, você pode também precisar de um software tolerante a defeitos. Existem duas abordagens para fornecer software tolerante a defeitos (figuras 20.11 e 20.12). Ambas as técnicas foram derivadas do modelo de hardware em que componentes redundantes (ou, talvez, sistemas redundantes) são incluídos e componentes defeituosos podem ser desativados.

As duas abordagens para tolerância a defeitos de software são:

1. *Programação em N-versões*. Usando uma especificação comum, o sistema de software é implementado em uma série de versões por diferentes equipes. Essas versões são executadas paralelamente em computadores separados. Suas saídas são comparadas com a utilização de um sistema de votação, e saídas inconsistentes ou saídas não produzidas em tempo são rejeitadas. No mínimo três versões de sistema devem ser disponibilizadas de modo que duas versões sejam consistentes no evento de uma única falha. Essa é a abordagem mais comum usada para tolerância a defeitos de software. Ela foi usada em sistemas de sinalização de ferrovias, em sistemas de aviação e em sistemas de proteção de reatores. Avizienis (Avizienis, 1985; Avizienis, 1995) descreve essa abordagem.

Figura 20.10

Redundância modular tripla para lidar com falha de hardware.

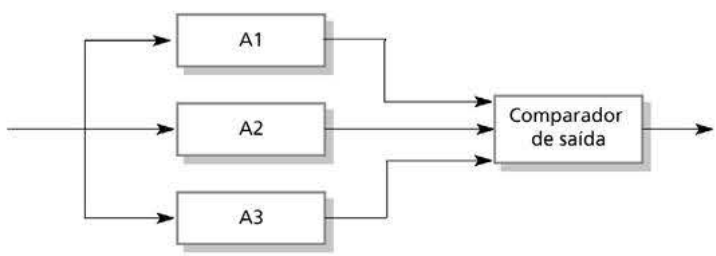


Figura 20.11
Programação em N-versões.

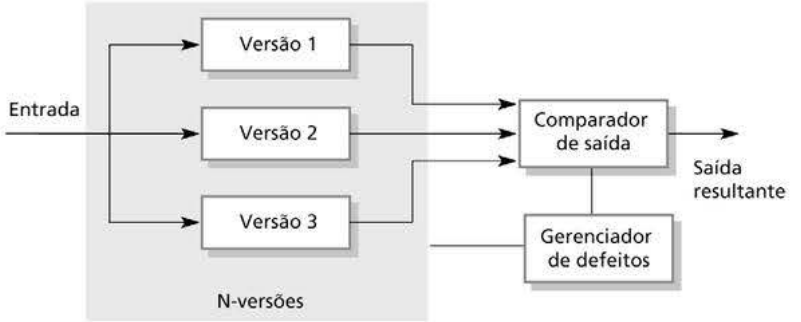
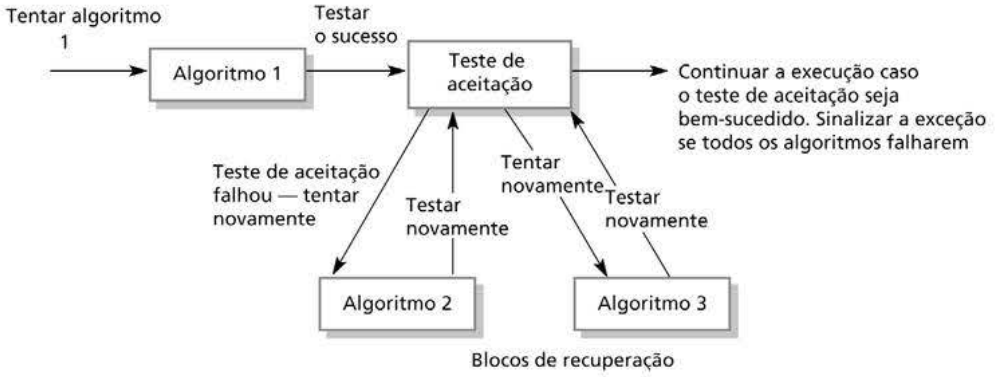


Figura 20.12
Blocos de recuperação.



2. *Blocos de recuperação*. Nessa abordagem, cada componente do programa inclui um teste para verificar se o componente foi executado com sucesso. Também inclui um código alternativo que permite ao sistema fazer uma cópia e repetir o processamento se o teste detectar uma falha. As implementações são, deliberadamente, interpretações diferentes da mesma especificação. Elas são mais executadas em seqüência do que em paralelo; dessa maneira o hardware replicado não é necessário. Na programação em N-versões, as implementações podem ser diferentes, mas não é incomum que duas ou mais equipes escolham o mesmo algoritmo para implementar a especificação. Randell (Randell, 1975) e Randell e Xu (Randell e Xu, 1995) descrevem o método de blocos de recuperação.

O fornecimento de software tolerante a defeitos requer que o software seja executado sob o controle de um controlador tolerante a defeitos que assegurará que as etapas envolvidas na tolerância a defeitos sejam executadas. Esse controlador examina as saídas e as compara. Se elas diferirem, algumas ações de recuperação poderão ser iniciadas. Laprie et al. (Laprie, et al., 1995) descrevem arquiteturas de sistemas tolerantes a defeitos.

Ambas as abordagens de tolerância a defeitos fazem uso de diversidade de projeto e implementação. Quando abordagens diversas são usadas para realizar a mesma implementação, é razoável supor que as diferentes versões de software não incluam os mesmos defeitos; sendo assim, falhas em comum são improváveis. A diversidade pode ser obtida de várias formas:

1. Incluindo-se requisitos para que diferentes abordagens para o projeto sejam usadas. Por exemplo, pode ser solicitado que uma equipe produza um projeto orientado a objetos e uma outra produza um projeto orientado a funções.
2. Solicitando que as implementações sejam escritas em linguagens de programação diferentes. Por exemplo, em um sistema de três versões, Ada, C++ e Java poderiam ser usadas para escrever as versões do software.
3. Solicitando o uso de ferramentas e de ambientes de desenvolvimento diferentes para o sistema.
4. Solicitando explicitamente algoritmos diferentes a serem usados em algumas partes da implementação. Entretanto, isso limita a liberdade da equipe de projeto e pode ser difícil reconciliar os requisitos de desempenho do sistema.

Cada equipe de desenvolvimento deve trabalhar com uma especificação de sistema — o V-spec — derivada da especificação de requisitos de sistema (Avizienis, 1995). Assim como na especificação de funcionalidade de sistema, o V-spec deve definir onde as saídas do sistema devem ser geradas para comparação. As equipes de desenvolvimento para cada versão devem trabalhar isoladamente para reduzir a probabilidade de desenvolverem erros em comum sobre o sistema.

A diversidade de projeto certamente aumenta a confiabilidade global do sistema. Contudo, uma série de experimentos sugeriu que a suposição de que equipes de projeto independentes não cometem os mesmos erros pode não ser sempre válida (Knight e Leveson, 1986; Brilliant, et al., 1990; Leveson, 1995). Equipes de desenvolvimento podem cometer os mesmos

erros devido a erros comuns de interpretação da especificação ou devido a eles chegarem independentemente aos mesmos algoritmos para resolver o problema. Blocos de recuperação reduzem a probabilidade de defeitos em comum porque algoritmos diferentes são usados explicitamente para cada bloco de recuperação.

O ponto fraco de ambas as abordagens para a tolerância a defeitos é que elas baseiam-se na premissa de que a especificação está correta. Elas não toleram erros de especificação. Em muitos casos, no entanto, a especificação está incorreta ou incompleta, por isso o sistema se comporta de maneira inesperada. Um modo de reduzir a possibilidade de erros de especificação em comum é desenvolver os V-specs para o sistema independentemente e definir as especificações em linguagens diferentes. Uma equipe de desenvolvimento poderia trabalhar com base em uma especificação formal; outra, a partir de um modelo de sistema baseado em estados; e, uma terceira, baseada em uma especificação em linguagem natural. Isso ajuda a evitar alguns erros de interpretação de especificação, mas não contorna o problema de defeitos de especificação.

PONTOS-CHAVE

- A confiança em um programa pode ser obtida evitando-se a introdução de defeitos pela detecção e remoção de defeitos antes da implantação do sistema e pela inclusão de recursos de tolerância a defeitos que permitem ao sistema permanecer em operação depois que um defeito tenha causado falha no sistema.
- O uso de redundância e de diversidade nos processos e nos sistemas de software é essencial para o desenvolvimento de sistemas confiáveis.
- O uso de um processo bem definido e repetível é importante para os defeitos de um sistema serem minimizados. O processo deve incluir atividades de verificação e validação em todos os estágios, desde a definição de requisitos até a implementação do sistema.
- Algumas construções e técnicas de programação, como comandos *goto*, ponteiros, recursão, herança e números em ponto flutuante, são inerentemente propensas a erros. Não devem ser usadas no desenvolvimento de sistemas confiáveis.
- As exceções são usadas para apoiar o gerenciamento de erros em sistemas confiáveis. Todas as exceções devem ser explicitamente tratadas em um sistema confiável.
- Os quatro aspectos de tolerância a defeitos de programa são: detecção de falhas, avaliação de danos, recuperação e reparação de defeitos.
- Programação em N-versões e blocos de recuperação são abordagens alternativas para arquiteturas tolerantes a defeitos, nas quais cópias redundantes do hardware e do software são mantidas. Ambas baseiam-se na diversidade de projeto e no uso de um controlador tolerante a defeitos para coordenar a execução de unidades redundantes de programa.

LEITURAS SUGERIDAS

Software fault tolerance techniques and implementation. Explicação abrangente das técnicas para se obter tolerância a defeitos de software e arquiteturas tolerantes a defeitos. O livro também aborda tópicos gerais sobre confiabilidade de software. (L. L. Pullum, 2001, Artech House.)

Handbook of software reliability engineering. Essa coleção inclui vários artigos que explicam blocos de recuperação e programação em N-versões. Também inclui um bom artigo sobre arquiteturas de sistemas tolerantes a defeitos. (M. R. Lyu (ed.), 1996, McGraw-Hill.)

EXERCÍCIOS

20.1 Forneça quatro motivos pelos quais raramente justifica-se o custo de as empresas assegurarem que seu software é livre de defeitos.

20.2 Forneça dois exemplos de atividades diferentes e redundantes que podem ser incorporadas a processos confiáveis.

20.3 Explique por que a herança é uma estrutura potencialmente propensa a erros e por que seu uso deve ser minimizado quando se desenvolvem sistemas críticos em uma linguagem orientada a objetos.

20.4 Explique os problemas de desenvolvimento e manutenção de sistemas 'nonstop', como o software de comutação telefônica. Como as exceções podem ser usadas no desenvolvimento de tais sistemas?

20.5 Explique por que você deve controlar explicitamente todas as exceções em um sistema tolerante a defeitos.

20.6 Descreva brevemente as estratégias de recuperação de erros forward e backward. Por que a recuperação de erros backward é usada com mais freqüência do que a recuperação forward? Forneça dois exemplos de classes de sistemas em que a recuperação de erros backward pode ser usada.

20.7 O que é essencial para uma recuperação de erros forward ser implementada em um sistema tolerante a defeitos? A recuperação de erros forward é possível em sistemas interativos?

20.8 Elabore um tipo abstrato de dados ou uma classe de objeto chamada **RobustList** que implemente a recuperação de erros forward em uma lista vinculada. Você deve incluir operações para verificar a lista quanto à possibilidade de corrompimento e reconstruí-la caso algum corrompimento tenha ocorrido. Suponha que você possa verificar o corrompimento mantendo referências forward e backward para e a partir dos membros adjacentes da lista.

20.9 Sugira circunstâncias em que seja apropriado o uso de uma arquitetura tolerante a defeitos ao se implementar um sistema de controle baseado em software, e explique por que essa abordagem é necessária.

20.10 Foi sugerido que o software de controle para uma máquina de tratamento com radioatividade (usada para tratar pacientes com câncer) deve ser implementado com o uso da programação em N-versões. Comente se você considera essa uma boa sugestão.

20.11 Forneça dois motivos pelos quais todas as versões de sistema em um sistema em N-versões podem falhar de maneira similar.

20.12 Usar as técnicas aqui explicadas para produzir software seguro obviamente envolve custos extras consideráveis. Que custos extras podem ser justificados se cem vidas forem salvas durante os 15 anos de tempo de vida de um sistema? Seriam justificados os mesmos custos se dez vidas forem salvas? Quanto vale uma vida? Os rendimentos das pessoas afetadas fazem a diferença nesse julgamento?

21

Evolução de software

Objetivos

Os objetivos deste capítulo são apresentar a evolução de software e descrever uma série de maneiras de como modificar um software. Após ler este capítulo, você:

- compreenderá que as mudanças são inevitáveis para que os sistemas de software permaneçam úteis e que o desenvolvimento e a evolução do software podem ser integrados em um modelo em espiral;
- terá aprendido sobre diferentes tipos de manutenção de software e fatores que afetam os custos de manutenção;
- estará ciente dos processos envolvidos na evolução de software, entre eles o processo de reengenharia de software;
- compreenderá como os sistemas legados podem ser avaliados para decidir se devem ser descartados, mantidos, sofrer reengenharia ou ser substituídos.

Conteúdo

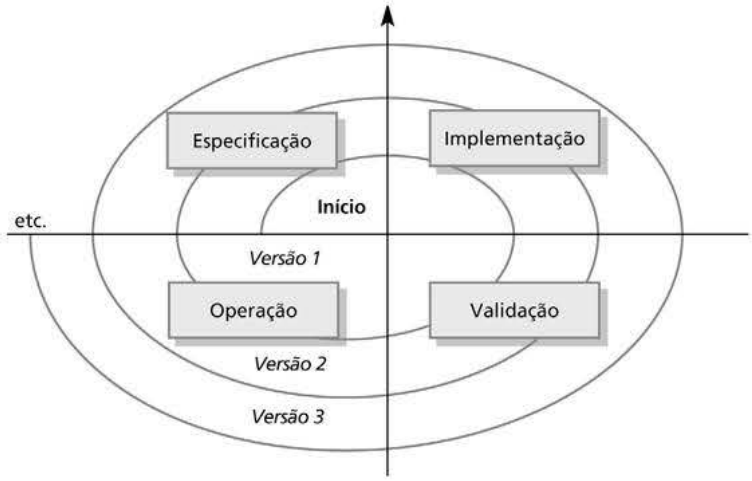
21.1 Dinâmica da evolução de programas
21.2 Manutenção de software
21.3 Processo de evolução
21.4 Evolução de sistemas legados

Depois de os sistemas terem sido implantados, eles devem inevitavelmente mudar para permanecerem úteis. Após o software ser colocado em uso, novos requisitos surgem e os requisitos existentes mudam. Mudanças nos negócios freqüentemente geram novos requisitos para o software existente. Partes do software podem ser modificadas para corrigir erros descobertos durante sua operação, para adaptá-lo a uma nova plataforma e para aprimorar seu desempenho e outras características não funcionais. O desenvolvimento de software, portanto, não pára quando um sistema é entregue para operação, mas prossegue ao longo do ciclo de vida do sistema.

A evolução do software é importante pois as organizações atuais são muito dependentes dos sistemas nos quais investiram milhões de dólares. Seus sistemas são importantes ativos de negócios e deve haver investimentos em mudanças para que o valor dos ativos seja mantido. A maior parte do orçamento nas grandes empresas é destinada à manutenção de sistemas existentes, e não devemos ficar surpresos com situações como as mostradas por Erlikh (Erlikh, 2000), que sugere que 90% dos custos de software estão na evolução. Existe grande variação nesse percentual, contudo, uma vez que as pessoas distinguem os custos de evolução dos de manutenção.

Como explico mais adiante, as mudanças pós-implantação não estão relacionadas simplesmente com reparos de defeitos no software. A maioria das mudanças é conseqüência de novos requisitos gerados em resposta às mudanças nos negócios ou necessidades dos usuários. Conseqüentemente,

Figura 21.1
Modelo em espiral de desenvolvimento e evolução.



pode-se pensar na engenharia de software como um processo em espiral com requisitos, projeto, implementação e teste caminhando juntos durante todo o ciclo de vida do sistema. Isso é ilustrado na Figura 21.1. Você deve começar criando o release 1 do sistema. Uma vez entregue, mudanças são propostas e o desenvolvimento do release 2 começa quase imediatamente. De fato, a necessidade de evolução se torna óbvia mesmo antes de o software ser implantado, de modo que releases posteriores do software podem estar em desenvolvimento antes que a versão inicial tenha sido liberada.

Esse é um modelo idealizado de evolução de software que pode ser aplicado em situações em que uma única organização é responsável tanto pelo desenvolvimento inicial do software como pela evolução dele. Os produtos de software mais genéricos são desenvolvidos usando-se essa abordagem. Por outro lado, o software sob encomenda pode ser desenvolvido externamente, mas sua evolução é de responsabilidade da equipe de desenvolvimento do cliente. Como alternativa, o usuário do software pode assinar um contrato separado com uma empresa externa para apoio e evolução do sistema.

Nesse caso, muitas vezes existem interrupções no processo em espiral. Documentos de requisitos e de projeto podem não ser passados de uma empresa para a outra. As empresas podem mesclar ou reorganizar, e herdar software de outras empresas para, então, descobrir que o software deve ser alterado. Quando a transição do desenvolvimento para a evolução é contínua, o processo de mudança do software depois da entrega freqüentemente é denominado *manutenção de software*. Conforme explicarei mais adiante neste capítulo, a manutenção envolve atividades extras de processo, como compreensão de programas, além das atividades normais de desenvolvimento de software.

21.1 Dinâmica da evolução de programas

A dinâmica da evolução de programas é o estudo de mudanças do sistema. A maioria dos trabalhos nessa área foi conduzida por Lehman e Belady, inicialmente nas décadas de 1970 e 1980 (Lehman e Belady, 1985). O trabalho continuou em 1990, quando Lehman e outros investigaram a importância do feedback em processos evolutivos (Lehman, 1996; Lehman, et al., 1998; Lehman, et al., 2001). Com base nesses estudos, propuseram um conjunto de leis (Leis de Lehman) relacionadas à mudança do sistema. Eles defendem que essas leis (hipóteses, na verdade) são invariáveis e amplamente aplicáveis. Lehman e Belady examinaram o crescimento e a evolução de um número considerável de sistemas de software de grande porte. As leis propostas, mostradas na Tabela 21.1, foram derivadas dessas medições.

A primeira lei estabelece que a manutenção do sistema é um processo inevitável. À medida que o ambiente do sistema muda, novos requisitos surgem e o sistema deve ser modificado. Quando o sistema modificado é novamente introduzido no ambiente, ocorrem mais mudanças ambientais e, assim, o ciclo reinicia o processo de evolução.

A segunda lei estabelece que, quando o sistema é alterado, sua estrutura é degradada. A única maneira de evitar isso é investir em manutenções preventivas nas quais são empregadas horas na aprimoração da estrutura do software sem adicionar novas funcionalidades. Obviamente, isso implica custos adicionais, além daqueles para implementar as mudanças necessárias do sistema.

A terceira lei é, talvez, a mais interessante e polêmica das leis de Lehman. Ela sugere que os sistemas de grande porte têm dinâmica própria definida ainda nos estágios iniciais do processo de desenvolvimento. Isso determina as duras tendências de manutenção do sistema e limita o número de possíveis mudanças. Lehman e Belady sugerem que essa lei é uma conseqüência de fatores estruturais que influenciam e restringem as mudanças do sistema, bem como de fatores organizacionais que afetam a evolução do sistema.

Tabela 21.1 Leis de Lehman

Lei	Descrição
Mudança contínua	Um programa usado em um ambiente real deve mudar necessariamente ou tornar-se progressivamente menos útil.
Complexidade crescente	À medida que um programa muda, sua estrutura tende a se tornar mais complexa. Recursos extras devem ser dedicados para preservar e simplificar a estrutura.
Evolução de programa de grande porte	A evolução de programa é um processo auto-regulável. Atributos de sistemas como tamanho, tempo entre versões e número de erros reportados é quase invariável em cada versão de sistema.
Estabilidade organizacional	Durante o ciclo de vida de um programa, sua taxa de desenvolvimento é quase constante e independente de recursos dedicados ao desenvolvimento do sistema.
Conservação de familiaridade	Durante o ciclo de vida de um sistema, mudanças incrementais em cada versão são quase constantes.
Crescimento contínuo	A funcionalidade oferecida pelos sistemas deve aumentar continuamente para manter a satisfação do usuário.
Qualidade em declínio	A qualidade dos sistemas entrará em declínio a menos que eles sejam adaptados a mudanças em seus ambientes operacionais.
Sistema de feedback	Os processos de evolução incorporam sistemas de feedback com vários agentes e loops e você deve tratá-los como sistemas de feedback para conseguir aprimoramentos significativos de produto.

Quando o sistema ultrapassa determinado tamanho ele se torna difícil de alterar. Como é grande e complexo, o sistema é difícil de ser compreendido, e os programadores ficam mais propensos a cometer erros e introduzir defeitos no sistema. Assim, as pequenas mudanças evitam a diminuição da confiabilidade do sistema. Uma grande mudança no sistema provavelmente introduzirá muitos novos defeitos que limitarão a utilidade da mudança entregue na nova versão do sistema.

Sistemas de grande porte são produzidos por grandes organizações com uma burocracia interna que define os orçamentos de mudanças para cada sistema e controla os processos de decisão dessas mudanças. As organizações devem tomar decisões com base em riscos, valores das mudanças e custos envolvidos. Tais decisões tomam tempo. Durante esse tempo, outras mudanças, de maior prioridade, podem ser propostas. Pode ser necessário deixar as mudanças originais para mais tarde. Os processos de tomada de decisões das organizações ditam, portanto, a taxa de mudanças do sistema.

A quarta lei de Lehman sugere que a maioria dos grandes projetos de programação acontece no que ele denomina estado *saturado*. Ou seja, uma mudança de recursos ou de pessoal tem efeitos imperceptíveis na evolução do sistema no longo prazo. Isso é consistente com a terceira lei, que sugere que a evolução do programa é independente das decisões gerenciais. Essa lei confirma que as equipes de desenvolvimento de grandes sistemas de software são muitas vezes improdutivas porque os overheads de comunicação dominam o trabalho da equipe.

A quinta lei de Lehman está relacionada aos incrementos de mudanças em cada release do sistema. Ao adicionar novas funcionalidades ao sistema, inevitavelmente são introduzidos novos defeitos. Quanto mais funcionalidades são introduzidas, mais defeitos são inseridos. Assim, um grande incremento na funcionalidade de um release do sistema significa que haverá um release posterior na qual os novos defeitos serão reparados. Relativamente, pouca funcionalidade será adicionada nesse release. A lei sugere que você não deve orçar grandes incrementos de funcionalidade sem levar em conta a necessidade de correção dos defeitos.

As cinco primeiras leis foram as propostas iniciais de Lehman; as demais leis foram adicionadas por outros trabalhos. A sexta e a sétima leis são similares e indicam essencialmente que os usuários se tornarão cada vez mais insatisfeitos com o sistema, a menos que ele seja atualizado e novas funcionalidades sejam adicionadas. A última lei reflete o mais recente trabalho em processos de feedback, embora não fique claro como pode ser aplicada de maneira prática no desenvolvimento de software.

As observações de Lehman geralmente são coerentes. Elas devem ser levadas em conta no planejamento do processo de manutenção. Pode acontecer, algumas vezes, de as observações serem ignoradas devido às considerações de negócios. Por exemplo, por razões de mercado, pode ser necessário fazer muitas e grandes mudanças num único release. As prováveis conseqüências disso são que um ou mais releases para correção de erros vão ser necessários.

Pode parecer que diferenças fundamentais óbvias entre releases de produtos de software violam as leis de Lehman. Por exemplo, o Microsoft Word foi transformado de um simples processador de texto que operava em 256 K de memória num

sistema gigantesco, carregado de recursos. Agora necessita de muitos megabytes de memória e de um processador mais rápido para funcionar. Sua evolução parece contradizer a quarta e a quinta leis de Lehman. Contudo, suspeito que esse programa não seja exatamente uma seqüência de revisões de um programa de núcleo comum. Na verdade, o nome foi mantido por razões de mercado e o programa foi reescrito e reestruturado mais de uma vez desde o lançamento de seu release original.

21.2 Manutenção de software

A manutenção de software é um processo geral de mudanças de um sistema depois que ele é entregue. O termo geralmente é aplicado ao software sob encomenda no qual grupos de desenvolvimento separados são envolvidos antes e depois da entrega do sistema. As mudanças feitas no software podem ser mudanças simples para corrigir erros de codificação, podem ser mudanças mais extensas para corrigir erros de projetos ou melhorias significativas para corrigir erros de especificação ou para acomodar novos requisitos. As mudanças são implementadas por meio de modificações de componentes de sistemas existentes e, quando necessário, por meio da adição de novos componentes ao sistema.

Existem três tipos diferentes de manutenção de software:

1. *Manutenção para reparo de defeitos de software*. A correção de erros de codificação é normalmente barata; os erros de projetos são mais caros, pois podem envolver a reescrita de vários componentes dos programas. Erros de requisitos são mais onerosos para serem reparados, pois pode ser necessário o reprojeto do sistema existente.
2. *Manutenção para adaptar o software a um ambiente operacional diferente*. Esse tipo de manutenção é necessária quando algum aspecto do ambiente do sistema, como o hardware, plataforma do sistema operacional ou outro software de apoio mudam. O sistema da aplicação deve ser modificado para lidar com essas mudanças de ambiente.
3. *Manutenção para adicionar funcionalidade ao sistema ou modificá-la*. Esse tipo de manutenção é necessária quando os requisitos do sistema mudam em resposta às mudanças organizacionais ou de negócios. A escala de mudanças necessárias para o software é muito maior do que em outros tipos de manutenção.

Na prática, não fica muito clara a distinção entre esses tipos de manutenção. Quando você adapta o sistema a um novo ambiente, isso pode ser feito pela adição de funcionalidade para aproveitar as características do novo ambiente. Os defeitos de software freqüentemente são expostos porque os usuários usam o sistema de maneiras imprevisíveis. Mudanças no sistema para acomodar o modo de trabalho é o melhor meio de corrigir esses defeitos.

Esses tipos de manutenção são bastante conhecidos, mas diferentes pessoas algumas vezes dão nomes diferentes a eles. O termo *manutenção corretiva* é universalmente usado para se referir à manutenção de reparo de defeitos. Contudo, *manutenção adaptativa* algumas vezes se refere à adaptação a um novo ambiente e pode significar adaptação do software aos novos requisitos. *Manutenção evolutiva* pode significar aperfeiçoar o software do sistema por meio da implementação de novas funcionalidades, mas em outras situações pode significar manter a funcionalidade do sistema, melhorando porém sua estrutura e seu desempenho. Devido a essas variações de nomenclatura, evitei o uso de todos esses termos no restante deste capítulo.

Pesquisas feitas por Lientz e Swanson (Lientz e Swanson, 1980) e Nosek e Palvia (Nosek e Palvia, 1990) sugerem que aproximadamente 65% das manutenções estão relacionadas à implementação de novos requisitos, 18% a mudanças do sistema para adaptá-lo a um novo ambiente operacional e 17% a corrigir defeitos do sistema (Figura 21.2). Para sistemas sob encomenda, a distribuição de custos ainda é correta. O ponto importante não é o percentual específico, mas o fato de que o reparo de defeitos do sistema não é uma atividade de manutenção onerosa. Promover a evolução do sistema para lidar com novos ambientes e os novos requisitos ou a alteração destes consomem mais esforço de manutenção.

Os custos de manutenção como parte dos custos de desenvolvimento variam de um domínio de aplicação para outro. Guimarães (Guimarães, 1983) sugere que os custos de manutenção de sistemas de aplicação de negócios são bastante comparáveis aos custos de desenvolvimento de sistemas. Para sistemas de tempo real incorporados, os custos de manutenção podem ser até quatro vezes mais altos do que os custos de desenvolvimento. Os requisitos de alta robustez e desempenho desses sistemas fazem com que seus módulos tenham de ser muito acoplados e, por isso, difíceis de serem modificados.

Geralmente é benéfico investir esforço no projeto e na implementação de um sistema para reduzir os custos de manutenção. Adicionar novas funcionalidades depois da entrega do sistema é oneroso porque você precisa empregar esforços para comprender o sistema e analisar o impacto das mudanças propostas. Assim, o trabalho executado durante o desenvolvimento para tornar o software mais fácil de ser compreendido e alterado contribui para reduzir os custos de manutenção. Boas técnicas de engenharia de software, como especificações precisas, uso da orientação a objetos no desenvolvimento e gerenciamento de configurações, contribuem para a redução dos custos de manutenção.

A Figura 21.3 mostra como os custos de todo o ciclo de vida podem diminuir quando mais esforço for despendido durante o desenvolvimento para produzir um sistema fácil de manter. Devido ao potencial de redução de custos na compre-

Figura 21.2
Distribuição de esforços de manutenção.

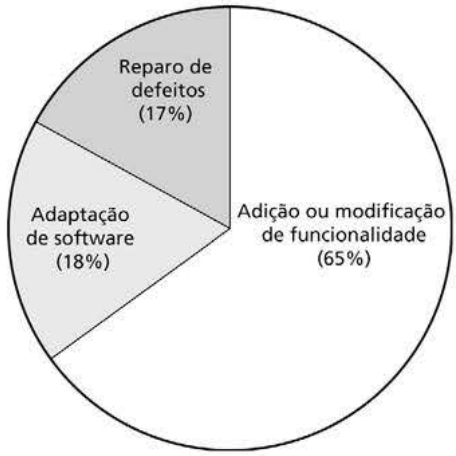
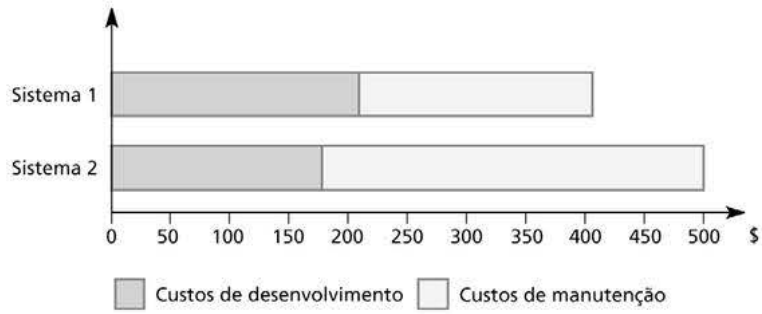


Figura 21.3
Custos de desenvolvimento e de manutenção.



ensão, análise e teste, existe um efeito multiplicador significativo quando o sistema é desenvolvido considerando-se a sua facilidade de manutenção. Para o Sistema 1, o custo extra de desenvolvimento de 25 mil dólares é investido para tornar o sistema mais fácil de manter. Isso resulta em uma economia de 100 mil dólares em custos de manutenção durante todo o ciclo de vida do software. Supõe-se que o percentual de aumento nos custos de desenvolvimento resulta em um decréscimo percentual comparável em custos globais do sistema.

Uma razão importante de os custos de manutenções serem altos é que é mais oneroso adicionar funcionalidade depois que o sistema está em operação do que a implementação da mesma funcionalidade durante o desenvolvimento. Os fatores fundamentais que distinguem o desenvolvimento e a manutenção e que levam a maiores custos de manutenção são:

1. *Estabilidade da equipe*. Depois que um sistema foi entregue, é normal que a equipe seja desfeita e as pessoas trabalhem em novos projetos. A nova equipe ou as pessoas responsáveis pela manutenção dos sistemas não entendem o sistema ou a base das decisões de projeto aplicadas ao sistema. Muito esforço durante o processo de manutenção é despendido para compreender o sistema existente antes de implementar as mudanças.
2. *Responsabilidade contratual*. O contrato para manter um sistema geralmente é separado do contrato de desenvolvimento deste. O contrato de manutenção pode ser concedido a uma empresa diferente da que desenvolveu o sistema. Esse fator, junto com a falta de estabilidade da equipe, indica que não há incentivo para que uma equipe de desenvolvimento escreva um software fácil de ser modificado. Se puderem ser feitas simplificações para poupar esforço durante o desenvolvimento, a equipe fará isso, mesmo acarretando aumento nos custos de desenvolvimento.
3. *Habilidade do pessoal*. O pessoal de manutenção geralmente é inexperiente e sem familiaridade com o domínio da aplicação. A manutenção tem uma imagem ruim entre os engenheiros de software. Ela é vista como um processo que exige menos habilidade do que o desenvolvimento de sistema e, muitas vezes, é alocado um pessoal menos experiente. Além disso, sistemas antigos podem ter sido escritos em linguagens de programação obsoletas. O pessoal de manutenção pode não ter tanta experiência em desenvolvimento nessas linguagens e devem aprendê-las para executar a manutenção do sistema.
4. *Idade e estrutura do programa*. À medida que os programas envelhecem, sua estrutura tende a se degradar pelas mudanças tornando-se, assim, cada vez mais difícil de ser compreendida e modificada. Alguns sistemas foram desenvolvidos sem as modernas técnicas de engenharia de software. Eles podem nunca ter sido bem estruturados e talvez

tenham sido otimizados levando-se em conta a eficiência em vez da facilidade de compreensão. A documentação do sistema pode estar perdida ou ser inconsistente. Sistemas antigos podem não ter sido submetidos ao gerenciamento de configuração, de modo que é desperdiçado muito tempo para encontrar as versões corretas dos componentes de sistema a serem modificados.

Os três primeiros problemas decorrem do fato de que muitas organizações ainda consideram o desenvolvimento e a manutenção como atividades separadas. A manutenção é vista como uma atividade de segunda classe, e não existe um incentivo para investimentos na redução dos custos de mudanças do sistema. A única solução de longo prazo para esse problema é aceitar que os sistemas raramente têm um ciclo de vida definido, porém continuam em uso, de alguma forma, por um período indefinido. Conforme sugeri na introdução, você deve pensar em sistemas do ponto de vista da evolução durante todo o seu ciclo de vida por meio de um processo de desenvolvimento contínuo.

O quarto item, o problema de estrutura degradada de sistema, é de certa maneira o problema mais fácil de se resolver. Técnicas de reengenharia (brevemente descritas mais adiante neste capítulo) podem ser aplicadas para aprimorar a estrutura e a facilidade de compreensão do sistema. Transformações de arquitetura podem adaptar o sistema a um novo hardware. Serviços de manutenção preventiva (essencialmente em reengenharia incremental) podem dar suporte para aprimorar o sistema e torná-lo mais fácil de ser modificado.

21.2.1 Previsão de manutenção

Os gerentes detestam surpresas, especialmente se elas resultam em altos custos inesperados. Você deve tentar prever quais mudanças de sistema são esperadas e quais partes do sistema são as mais difíceis de sofrerem manutenção. Você deve também tentar estimar os custos globais de manutenção para o sistema em dado período de tempo. A Figura 21.4 ilustra essas previsões e as questões associadas.

Essas previsões estão obviamente relacionadas a:

1. Se uma mudança de sistema deve ser aceita, isso depende, de certa maneira, da facilidade de manutenção dos componentes de sistema afetados por essa mudança.
2. A implementação de mudanças de sistema tende a degradar sua estrutura, reduzindo assim sua facilidade de manutenção.
3. Os custos de manutenção dependem do número de mudanças e os custos de implementação de mudanças dependem da facilidade de manutenção dos componentes do sistema.

A previsão do número de solicitações de mudança para um sistema requer uma compreensão do relacionamento entre o sistema e seu ambiente externo. Alguns sistemas têm um relacionamento bastante complexo com seu ambiente externo e mudanças nesse ambiente resultam em inevitáveis mudanças no sistema. Para avaliar o relacionamento entre o sistema e seu ambiente, você deve avaliar:

1. *O número e a complexidade das interfaces de sistema.* Quanto maior o número de interfaces e quanto mais complexas, mais prováveis serão as demandas quanto a mudanças.

Figura 21.4 Previsão de manutenção.

2. *O número de requisitos do sistema inerentemente voláteis.* Conforme expliquei no Capítulo 7, os requisitos que refletem políticas e procedimentos organizacionais são, provavelmente, mais voláteis do que os requisitos baseados em características de domínios estáveis.
3. *Os processos de negócios nos quais o sistema é usado.* À medida que os processos de negócio evoluem, eles geram solicitações de mudanças. Quanto mais os processos de negócio usam o sistema, mais demandas quanto a mudanças ocorrem.

Para prever a facilidade de manutenção do sistema, você deve entender a quantidade e os tipos de relacionamentos entre os componentes do sistema bem como a complexidade inerente a esses componentes. Há diversos estudos de diferentes tipos de complexidade em um sistema (McCabe, 1976; Halstead, 1977) e de relacionamentos entre complexidade e facilidade de manutenção (Kafura e Reddy, 1987; Banker, et al., 1993). Não é surpresa que esses estudos tenham descoberto que quanto mais complexo for um sistema ou um componente, mais onerosa será sua manutenção.

Medições de complexidade foram consideradas particularmente úteis na identificação individual de componentes do programa que são provavelmente os mais difíceis de serem mantidos. Kafura e Reddy (Kafura e Reddy, 1987) examinaram um número de componentes de sistema e descobriram que o esforço de manutenção tende a enfocar um número pequeno de componentes complexos. Eles sugerem que, para reduzir os custos de manutenção, você deve substituir, especificamente, componentes complexos do sistema por alternativas mais simples.

Depois de um sistema ter sido colocado em operação, você pode ser capaz de usar os dados de processo para ajudar a prever a facilidade de manutenção. Exemplos de métricas de processo que podem ser usadas para avaliar a facilidade de manutenção são:

1. *Número de solicitações de manutenção corretiva.* Um aumento no número de relatórios de falhas pode indicar que mais erros estão sendo introduzidos no programa do que reparados durante o processo de manutenção. Eles podem indicar um declínio na facilidade de manutenção.
2. *Tempo médio necessário e desejado para análise de impactos.* Isso reflete o número de componentes afetados pela solicitação de mudança. Se esse tempo aumentar, significa que cada vez mais componentes estão sendo afetados e a facilidade de manutenção está diminuindo.
3. *Tempo médio para implementar uma solicitação de mudança.* Não é igual ao tempo para análise de impactos embora haja uma correlação. Essa é a quantidade de tempo necessário para realmente modificar o sistema e sua documentação após a avaliação de quais componentes serão afetados. Um aumento no tempo necessário para implementar uma mudança indica um declínio na facilidade de manutenção.
4. *Número de solicitações pendentes de mudança.* Um aumento desse número com o tempo pode indicar declínio na facilidade de manutenção.

As informações de previsão sobre as solicitações de mudanças e sobre a facilidade de manutenção do sistema são usadas para prever os custos de manutenção. A maioria dos gerentes combina essas informações usando a intuição e a experiência para estimar os custos. O modelo de estimativa de custo COCOMO 2 (Boehm, et al., 2000), explicado no Capítulo 26, sugere que uma estimativa do esforço de manutenção de software pode ser baseada no esforço para compreender o código existente e no esforço para desenvolver um novo código.

21.3 Processo de evolução

Os processos de evolução de software variam consideravelmente, dependendo do software a ser mantido, dos processos de desenvolvimento usados e das pessoas envolvidas no processo. Em algumas organizações, a evolução pode ser um processo informal no qual as solicitações de mudanças, na maioria das vezes, provêm de conversas entre os usuários e desenvolvedores. Em outras empresas, é um processo formalizado, com documentação estruturada e produzida em cada estágio do processo.

Propostas de mudanças no sistema são os direcionadores da evolução de sistemas em todas as organizações. Essas propostas de mudanças podem envolver requisitos existentes ainda não implementados no sistema liberado, solicitações de novos requisitos ou reparos de defeitos detectados pelos stakeholders do sistema, além de novas idéias e propostas de aprimoramento de software solicitadas pela equipe de desenvolvimento. Conforme ilustrado na Figura 21.5, os processos de identificação de mudança e evolução são cíclicos e continuam ao longo do ciclo de vida de um sistema.

O processo de evolução inclui as atividades fundamentais de análise de mudanças, planejamento de releases, implementação de mudanças e liberação de um sistema para os clientes. O custo e o impacto dessas mudanças são avaliados para descobrir quanto do sistema é afetado pelas mudanças e quanto pode custar para implementar a mudança. Durante o

Figura 21.5
Processos de identificação de mudança e evolução.

planejamento de releases, todas as mudanças propostas (reparo de defeitos, adaptações e novas funcionalidades) são consideradas. Uma decisão é então tomada sobre quais mudanças devem ser implementadas na próxima versão do sistema. As mudanças são implementadas e validadas, e um novo release do sistema é liberado. O processo se repete com um novo conjunto de mudanças propostas para o próximo release. A Figura 21.6, adaptada de Arthur (Arthur, 1988), mostra uma visão geral desse processo.

O processo de implementação de mudanças é, essencialmente, uma iteração do processo de desenvolvimento na qual as revisões do sistema são projetadas, implementadas e testadas. Contudo, uma diferença importante é que o estágio inicial da implementação de mudanças é a compreensão do programa. Durante essa fase, você deve entender como o programa está estruturado e como ele entrega sua funcionalidade. Ao implementar uma mudança, você usa esse entendimento para assegurar que a mudança implementada não prejudicará o sistema existente.

De preferência, o estágio de implementação de mudanças desse processo deve modificar a especificação, o projeto e a implementação do sistema para refletir as mudanças no sistema (Figura 21.7). Novos requisitos que refletem as mudanças no sistema são propostos, analisados e validados. Os componentes de sistemas são reprojetados e implementados, e o sistema é retestado. Caso apropriado, pode ser realizada a prototipação das mudanças propostas como parte do processo de análise.

Quando ocorre uma mudança no software, são desenvolvidos sucessivos releases do sistema. Esses releases são compostos de novas versões de componentes do sistema. Você deve acompanhar as versões para garantir que as versões corretas dos componentes estão sendo usadas em cada release do sistema. O gerenciamento de configurações será abordado no Capítulo 29.

Durante o processo de evolução, os requisitos são analisados em detalhes e, freqüentemente, surgem implicações não aparentes no início do processo de análise das mudanças. Isso significa que as mudanças propostas podem ser alteradas e discussões posteriores com o cliente podem ser necessárias antes que as mudanças sejam implementadas.

Solicitações de mudanças muitas vezes estão relacionadas a problemas do sistema que precisam ser resolvidos urgentemente.

Figura 21.6 Processo de evolução de sistema.

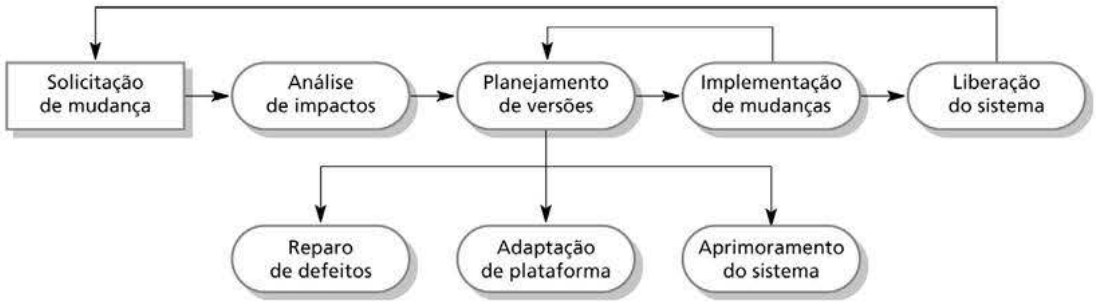
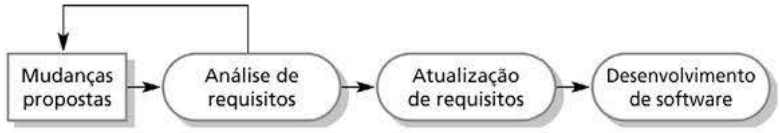


Figura 21.7
Implementação de mudança.



Essas mudanças urgentes podem surgir por três razões:

1. Ocorreu um defeito grave do sistema que deve ser reparado para permitir a continuidade da operação normal
2. Efeitos inesperados de mudanças no ambiente do sistema operacional que interrompem a operação normal
3. Ocorreram mudanças imprevistas na empresa que utiliza o sistema, como uma emergência provocada por novos concorrentes ou a introdução de uma nova legislação

Nesses casos, a necessidade de mudanças rápidas significa que pode não ser possível seguir o processo formal de análise de mudanças. Em vez de modificar os requisitos ou o projeto, você deve fazer uma correção de emergência no programa para resolver o problema imediato (Figura 21.8). Contudo, o perigo é que os requisitos, o projeto de software e a codificação se tornam gradativamente inconsistentes. Enquanto você está tentando documentar a mudança nos requisitos e no projeto, outras correções emergenciais no software podem ser necessárias. Elas têm prioridade sobre a documentação. Eventualmente, a mudança original é esquecida e a documentação e o código nunca se tornam consistentes.

Um outro problema com os reparos emergenciais de sistema é que eles precisam ser concluídos o mais rápido possível. Você escolhe uma solução de rápida implementação em vez da melhor solução do ponto de vista da estrutura do sistema. Isso acelera o envelhecimento do software de modo que as mudanças futuras se tornam progressivamente mais difíceis e os custos de manutenção aumentam.

Preferencialmente, quando são realizados reparos emergenciais de código, a solicitação da mudança deve permanecer com o status pendente após a correção dos defeitos do código. Ela poderá ser reimplementada depois de uma análise mais cuidadosa. Naturalmente, o código do reparo pode ser reusado. Uma solução alternativa e mais adequada para o problema pode ser descoberta quando houver mais tempo disponível para análise. Na prática, contudo, é quase inevitável que essas mudanças tenham baixa prioridade e, depois que outras mudanças de sistema forem feitas, não será realista refazer os reparos emergenciais.

21.3.1 Reengenharia de sistemas

Conforme explicado na seção anterior, o processo de evolução envolve a compreensão do programa a ser mudado e, depois, a implementação das mudanças. Contudo, muitos sistemas, especialmente os sistemas legados mais antigos (explicados no Capítulo 2), são difíceis de compreender e de mudar. Os programas podem ter sido originalmente otimizados para favorecer o desempenho ou utilização de espaço de memória à custa da facilidade de compreensão, ou, com o tempo, a estrutura inicial do programa pode ter sido corrompida por uma série de mudanças.

Para simplificar os problemas de mudança de sistemas legados, uma empresa pode decidir fazer a reengenharia desses sistemas para aprimorar sua estrutura e a facilidade de compreensão. A reengenharia de software está relacionada à reimplementação de sistemas legados para torná-los mais fáceis de manter. A reengenharia pode envolver uma nova documentação, organização e reestruturação do sistema, conversão do sistema em uma linguagem mais moderna e modificação e atualização da estrutura e dos valores dos dados de sistema. A funcionalidade do software não é alterada e, normalmente, a arquitetura do sistema também permanece a mesma.

A realização de reengenharia de um sistema de software tem duas vantagens fundamentais sobre as abordagens mais radicais de evolução de sistema:

1. *Risco reduzido*. Existe alto risco no redesenvolvimento de um software crítico de negócios. Podem ser cometidos erros nas especificações do sistema ou pode haver problemas de desenvolvimento. Atrasos na introdução do novo software podem significar a perda de negócios e acarretar custos extras. Por exemplo, em 1999, uma grande empresa norte-americana de alimentos atrasou a introdução do novo sistema de pedidos provocando atrasos nas entregas de produtos equivalentes a 100 milhões de dólares durante um pico na temporada de vendas.
2. *Custo reduzido*. O custo da reengenharia é significativamente menor do que o custo de desenvolvimento de um novo software. Ulrich (Ulrich, 1990) cita um exemplo de um sistema comercial em que os custos de reimplementação foram estimados em cerca de 50 milhões de dólares. O sistema passou por uma reengenharia bem-sucedida de 12 milhões de dólares. Suspeito que, com modernas tecnologias de software, o custo relativo de reimplementação será menor do que esse, mas ainda assim excederá consideravelmente os custos de reengenharia.

A distinção principal entre reengenharia e desenvolvimento de um novo software é o ponto de partida para o desenvolvimento. Em vez de começar o desenvolvimento com uma especificação escrita, o sistema antigo funciona como uma especificação para o novo sistema. Chikofsky e Cross (Chikofsky e Cross, 1990) denominaram o desenvolvimento convencional de

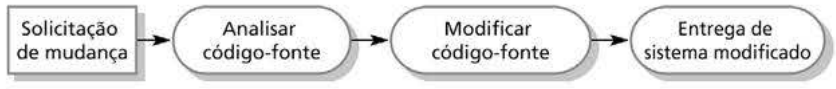
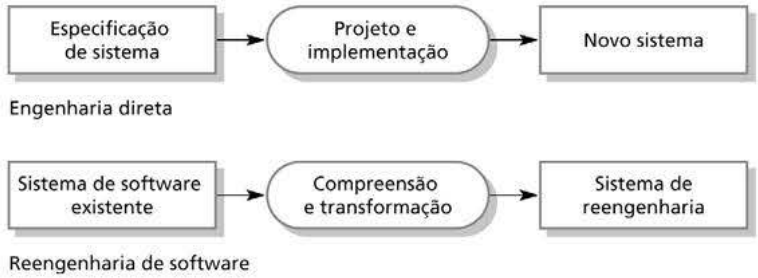


Figura 21.8
Processo de reparo de emergência.

Figura 21.9
Engenharia direta e reengenharia.



engenharia direta para distingui-lo da reengenharia de software. Essa distinção está ilustrada na Figura 21.9. A engenharia direta começa com uma especificação de sistema e envolve o projeto e a implementação do novo sistema. A reengenharia começa com um sistema existente, e o processo de desenvolvimento para a substituição baseia-se na compreensão e na transformação do sistema original.

A Figura 21.10 ilustra o processo de reengenharia. A entrada do processo é um programa legado e a saída é uma versão estruturada e modular do mesmo programa. Durante a reengenharia do programa, os dados do sistema podem também passar por reengenharia. As atividades nesse processo de reengenharia são:

1. *Conversão de código-fonte*. O programa é convertido de uma linguagem antiga de programação para uma versão mais moderna da mesma linguagem ou para uma linguagem diferente.
2. *Engenharia reversa*. O programa é analisado e são extraídas informações dele. Isso ajuda a documentar sua organização e funcionalidade.
3. *Aprimoramento da estrutura do programa*. A estrutura de controle do programa é analisada e modificada para facilitar a leitura e a compreensão.
4. *Modularização de programa*. As partes relacionadas do programa são agrupadas e, quando apropriado, as redundâncias são removidas. Em alguns casos, esse estágio pode envolver transformações de arquitetura em que um sistema centralizado, planejado para um único computador, é modificado para operar em uma plataforma distribuída.
5. *Reengenharia de dados*. Os dados processados são alterados para refletir as mudanças do programa.

A reengenharia de sistema pode não requerer necessariamente todos os passos da Figura 21.10. A conversão do código-fonte pode não ser necessária se a linguagem de programação usada para desenvolver o sistema ainda tiver suporte do fornecedor do compilador. Se a reengenharia conta totalmente com ferramentas automatizadas, a recuperação de documentação por meio de engenharia reversa pode ser desnecessária. A reengenharia de dados será necessária somente se as estruturas de dados mudarem durante a reengenharia do sistema. Contudo, a reengenharia de software sempre envolve alguma reestruturação de programa.

Para que um sistema que passou por reengenharia interopere com o novo software, pode ser necessário desenvolver componentes adaptadores, conforme explicado no Capítulo 19. Eles ocultam as interfaces originais do sistema de software e apresentam interfaces novas e mais estruturadas que podem ser usadas por outros componentes. Esse processo de empacotamento de sistemas legados é uma técnica importante para o desenvolvimento de componentes reusáveis em grande escala.

Figura 21.10 Processo de reengenharia.

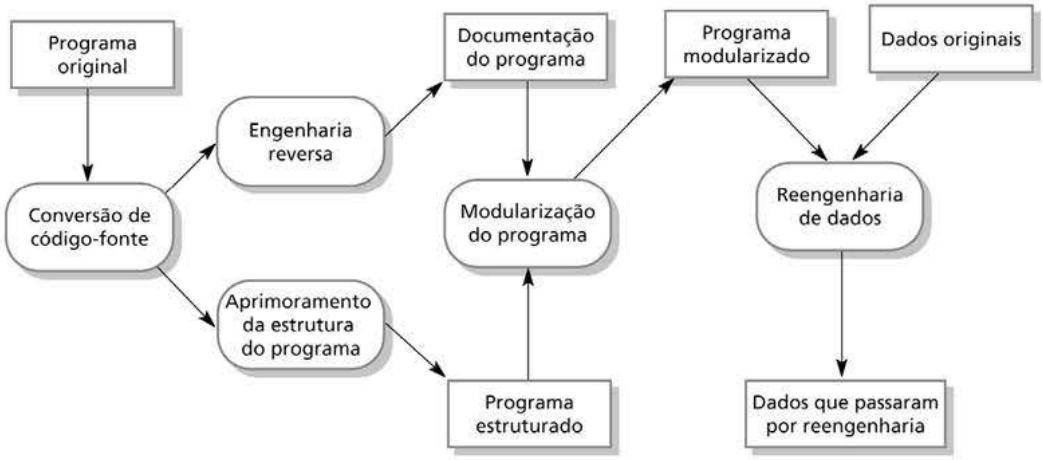
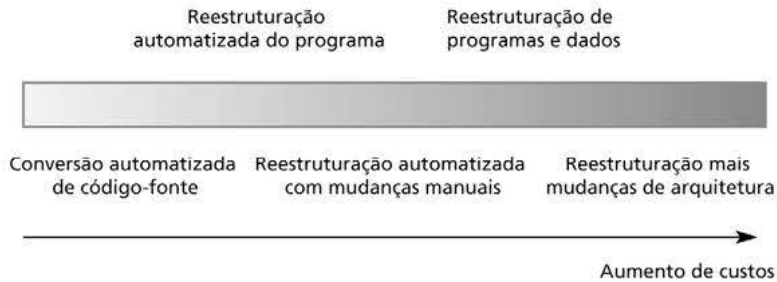


Figura 21.11
Abordagens de reengenharia.



Os custos da reengenharia dependem obviamente da extensão do trabalho que é realizado. Existe uma gama de abordagens possíveis para a reengenharia, conforme mostrado na Figura 21.11. Os custos aumentam da esquerda para a direita, sendo assim a conversão de código é a opção mais barata. A reengenharia, como parte da migração de arquiteturas, é mais dispendiosa.

Independente da extensão da reengenharia, os principais fatores que afetam os custos dela são:

1. *A qualidade do software que deve passar pela reengenharia.* Quanto menor a qualidade do software e de sua documentação associada (caso exista), maiores serão os custos de reengenharia.
2. *O apoio de ferramentas disponíveis para reengenharia.* Normalmente, não é adequado em termos de custo fazer a reengenharia de um sistema de software, a menos que você use ferramentas CASE para automatizar a maioria das mudanças.
3. *Extensão da conversão de dados.* Se a reengenharia exigir que grandes volumes de dados sejam convertidos, o custo do processo aumentará significativamente.
4. *A disponibilidade do pessoal especializado.* Se o pessoal responsável pela manutenção do sistema não puder ser envolvido no processo de reengenharia, os custos aumentarão, pois os engenheiros de sistema responsáveis pela reengenharia terão de empregar muito tempo para a compreensão do sistema.

A principal desvantagem da reengenharia de software é que existem limitações práticas referentes a quanto um sistema pode ser aprimorado. Não é impossível, por exemplo, converter um sistema escrito por meio de uma abordagem funcional em um sistema orientado a objetos. Grandes mudanças de arquitetura ou reorganizações radicais do gerenciamento de dados do sistema não podem ser realizadas automaticamente e, assim, incorrem em altos custos adicionais. Embora a reengenharia possa aprimorar a facilidade de manutenção, um sistema que passou por reengenharia provavelmente não será mais fácil de manter que um novo sistema desenvolvido usando métodos modernos de engenharia de software.

21.4 Evolução de sistemas legados

Para novos sistemas desenvolvidos por meio de processos modernos de engenharia de software, como o desenvolvimento iterativo e CBSE, é possível planejar como integrar o desenvolvimento e a evolução de sistema. Cada vez mais empresas estão começando a compreender que o processo de desenvolvimento de sistema prossegue por todo o ciclo de vida e que uma separação artificial entre o desenvolvimento e a manutenção de software é inútil. Contudo, existem ainda muitos sistemas legados que são sistemas críticos de negócios. Esses sistemas precisam ser ampliados e adaptados às mudanças das práticas de e-business.

As organizações com um orçamento limitado para manter e atualizar seus sistemas legados precisam decidir como obter o melhor retorno sobre seu investimento. Isso significa que elas devem fazer uma avaliação realista de seus sistemas legados e, depois, decidirem qual é a estratégia mais apropriada para a evolução dos sistemas. Existem quatro opções de estratégia:

1. *Descartar o sistema completamente.* Essa opção deve ser escolhida quando o sistema não está mais contribuindo eficientemente com os processos de negócio. Isso ocorre quando os processos de negócio mudaram desde que o sistema foi instalado e não são mais completamente dependentes do sistema. Essa situação foi muito comum quando terminais de mainframe foram substituídos por PCs e softwares comerciais foram adaptados nessas máquinas para fornecer mais apoio computacional às necessidades do processo de negócio.
2. *Deixar o sistema sem alterações e continuar com a manutenção regular.* Essa opção deve ser escolhida quando o sistema é ainda necessário, mas bastante estável, e os usuários de sistema têm relativamente poucas solicitações de mudança.

3. *Reengenharia do sistema para aprimorar sua facilidade de manutenção.* Essa opção deve ser a escolhida quando a qualidade do sistema foi degradada por mudanças regulares e quando essas mudanças ainda são necessárias. Conforme expliquei, esse processo pode incluir o desenvolvimento de novos componentes de interface de modo que o sistema original possa operar com outros sistemas mais novos.
4. *Substituir todo ou parte do sistema por um novo sistema.* Essa opção deve ser escolhida quando outros fatores, como um novo hardware, impossibilitam que o sistema antigo possa continuar em operação ou quando sistemas comerciais permitem que um novo sistema seja desenvolvido a um custo razoável. Em muitos casos, a estratégia de substituição evolucionária pode ser adotada quando os componentes principais do sistema são substituídos por sistemas comerciais com outros componentes reusados sempre que possível.

Naturalmente, essas opções não são excludentes e, assim, quando um sistema é composto por vários programas, diferentes opções devem ser aplicadas a diferentes partes do sistema.

Ao avaliar um sistema legado, você deve observá-lo do ponto de vista dos negócios e da perspectiva técnica (Warren, 1998). Do ponto de vista de mercado, você precisa decidir se a empresa realmente precisa do sistema. Da perspectiva técnica, você precisa avaliar a qualidade do software da aplicação e o software e hardware de apoio do sistema. Você usa, portanto, uma combinação de valor de mercado e de qualidade do sistema para fundamentar sua decisão sobre o que fazer com o sistema legado.

Para ilustrar isso, vamos supor que uma organização tenha dez sistemas legados. A qualidade e o valor de mercado de cada um desses sistemas são avaliados e comparados com outros por meio de gráficos que mostram o valor relativo de mercado e a qualidade de sistema. Isso é mostrado na Figura 21.12.

Observando a Figura 21.12, você pode verificar que existem quatro agregações de sistema:

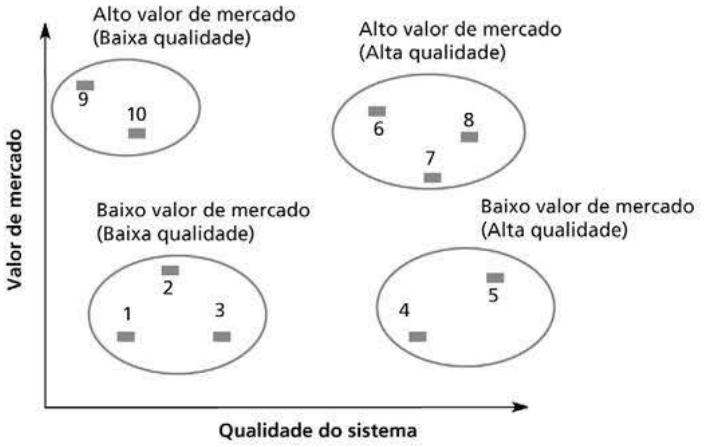
1. *Baixa qualidade, baixo valor de mercado.* A manutenção desses sistemas em operação será dispendiosa e a taxa de retorno para o negócio será bastante pequena. Esses sistemas devem ser descartados.
2. *Baixa qualidade, alto valor de mercado.* Esses sistemas contribuem de maneira importante para a empresa e não podem ser descartados. Contudo, sua baixa qualidade significa que é dispendioso mantê-los. Esses sistemas devem sofrer reengenharia para aprimorar sua qualidade ou serem substituídos caso um sistema comercial adequado esteja disponível.
3. *Alta qualidade, baixo valor de mercado.* Esses sistemas são os que não contribuem mais para a empresa, mas sua manutenção não é muito dispendiosa. Não vale a pensa substituí-los, de modo que as manutenções normais podem prosseguir desde que nenhuma mudança dispendiosa seja necessária e o hardware do sistema esteja operacional. Se mudanças dispendiosas forem necessárias, os sistemas deverão ser descartados.
4. *Alta qualidade, alto valor de mercado.* Esses sistemas devem ser mantidos em operação, mas sua qualidade significa que você não deve investir em transformação ou substituição. A manutenção normal deve prosseguir.

Para avaliar o valor de mercado de um sistema, você deve identificar os stakeholders do sistema, como os usuários finais e seus gerentes, e formular uma série de questões sobre o sistema. Existem quatro questões básicas que devem ser discutidas:

1. *Uso do sistema.* Se os sistemas são usados ocasionalmente ou por um pequeno número de pessoas, eles podem ter um baixo valor de mercado. Um sistema legado pode ter sido desenvolvido para atender a uma necessidade da empresa que mudou ou que agora pode ser atendida mais eficientemente de outras maneiras.

Figura 21.12

Avaliação de sistemas legados.



2. *Processos de mercado apoiados*. Quando um sistema é introduzido, podem ser projetados processos de mercado para explorar esse sistema. Contudo, as mudanças nesses processos podem ser impossíveis, pois o sistema legado não pode ser adaptado. Portanto, um sistema pode ter baixo valor de mercado porque impede a introdução de novos processos.
3. *A confiabilidade do sistema*. A confiabilidade do sistema não é apenas um problema técnico, mas também um problema de negócios. Se um sistema não for confiável e os problemas afetarem diretamente os clientes ou exigirem que pessoas sejam desviadas de suas tarefas para resolver esses problemas, o sistema tem baixo valor de mercado.
4. *As saídas do sistema*. A questão fundamental aqui é a importância das saídas do sistema para o funcionamento bem-sucedido da empresa. Se a empresa depende dessas saídas, o sistema tem um alto valor de mercado. Por outro lado, se essas saídas puderem ser facilmente geradas de alguma outra maneira ou se o sistema produzir saídas raramente usadas, seu valor de mercado pode ser baixo.

Por exemplo, vamos considerar que uma empresa forneça um sistema de pedidos de reservas de viagem no qual o pessoal responsável por organizar as viagens pode enviar pedidos por meio de agentes de viagem homologados. As passagens são entregues e a empresa é cobrada por elas. Contudo, uma avaliação de valor de mercado pode revelar que esse sistema é usado somente para uma porcentagem muito pequena de reservas de viagem. As pessoas que organizam as viagens consideram esse sistema mais barato e conveniente por lidar diretamente com os agentes de viagem por meio de sites Web. Esse sistema ainda pode ser usado, mas não existe uma justificativa real para mantê-lo. A mesma funcionalidade está disponível em sistemas externos.

Por outro lado, digamos que a empresa desenvolveu um sistema que mantém o acompanhamento de todos os pedidos anteriores de clientes e gera avisos automaticamente para que os clientes peçam os produtos novamente Isso resulta em altas taxas de pedidos repetidos e mantém os clientes satisfeitos porque eles percebem que o fornecedor está ciente de suas necessidades As saídas de tal sistema são muito importantes para a empresa; esse sistema tem, portanto, um alto valor de mercado.

Para avaliar um sistema de software com base em uma perspectiva técnica, você precisa considerar ambos, o próprio sistema da aplicação e o ambiente no qual o sistema opera. O ambiente inclui hardware e todo o apoio de software associado, como compiladores e linkers, necessário para manter o sistema. O ambiente é importante, pois muitas mudanças de sistema resultam de mudanças de ambiente, como atualizações de hardware ou sistemas operacionais.

Se possível, no processo de avaliação de ambiente, você deve realizar medições de sistema e de seus processos de manutenção. Exemplos de dados que podem ser úteis incluem custos de manutenção de sistema de hardware e de apoio de software, o número de defeitos de hardware que ocorrem em determinado período e a freqüência de patches e correções aplicadas ao software de apoio do sistema.

Os fatores a serem considerados durante a avaliação de ambientes são mostrados na Tabela 21.2. Observe que esses fatores não são todos características técnicas do ambiente. Você deve também considerar a confiabilidade dos fornecedores de hardware e de software de apoio. Se esses fornecedores não estão mais atuando no mercado, pode não haver apoio de manutenção para os sistemas.

Para avaliar a qualidade técnica de um sistema de aplicação, você deve avaliar vários fatores (Tabela 21.3) relacionados principalmente à confiabilidade do sistema, às dificuldades de manutenção do sistema e à documentação de sistema. Você pode também coletar dados quantitativos que ajudarão a julgar a qualidade do sistema. Exemplos de dados quantitativos que podem ser coletados são:

1. *O número de solicitações de mudança no sistema*. As mudanças tendem a corromper a estrutura do sistema e tornam as mudanças futuras mais difíceis. Quanto mais alto for esse valor, mais baixa será a qualidade do sistema.
2. *O número de interfaces com o usuário*. Este é um fator importante em sistemas baseados em formulários, em que cada formulário pode ser considerado como uma interface separada com o usuário. Quanto mais interfaces, maior é a probabilidade de inconsistências e redundâncias.
3. *O volume de dados usados pelo sistema*. Quanto maior o volume de dados (número de arquivos, tamanho do banco de dados etc.), mais complexo será o sistema.

Embora esses dados sejam muitas vezes úteis, colecioná-los pode ser muito dispendioso e, portanto, impraticável. Além disso, não existem valores absolutos a serem usados. A idade e o tamanho do sistema devem ser levados em conta nos julgamentos de qualidade baseados nessas medições.

De preferência, avaliações objetivas devem ser usadas para informar decisões sobre o que fazer com um sistema legado. Contudo, em muitos casos, essas decisões não são realmente objetivas, mas baseadas em considerações organizacionais e políticas. Por exemplo, se duas empresas se fundem, geralmente o parceiro mais poderoso politicamente irá manter seu sistema e descartar os demais. Se a gerência sênior de uma organização decide migrar para uma nova plataforma de

Tabela 21.2 Fatores usados na avaliação de ambientes

Fator	Questões
Estabilidade do fornecedor	O fornecedor ainda existe? O fornecedor é financeiramente estável e é provável que continue a existir? Se o fornecedor não estiver mais atuando no mercado, alguém mais mantém os sistemas?
Taxa de falhas	O hardware tem alta taxa de falhas relatadas? O software de apoio 'trava' ou força o reinício do sistema?
Idade	Qual é a idade do hardware e do software? Quanto mais antigos o hardware e o software de apoio, mais obsoletos eles serão. Eles podem ainda funcionar corretamente, mas pode haver benefícios econômicos e de mercado em migrar para sistemas mais modernos.
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm efeito significativo sobre os usuários do sistema?
Requisitos de apoio	Qual apoio local é exigido pelo hardware e pelo software? Se houver altos custos associados a esse apoio, pode valer a pena considerar a substituição do sistema.
Custos de manutenção	Quais são os custos de manutenção de hardware e de licenças de software de apoio? Hardwares mais antigos podem ter custos de manutenção maiores do que os de sistemas modernos. O software de apoio pode ter altos custos anuais de licença.
Interoperabilidade	Existem problemas de interface do sistema com outros sistemas? Os compiladores podem, por exemplo, ser usados com versões atuais do sistema operacional? É necessária a emulação de hardware?

Tabela 21.3 Fatores usados na avaliação de aplicações

Fator	Questões
Facilidade de compreensão	Qual é a dificuldade para compreender o código-fonte do sistema atual? Qual é a complexidade das estruturas de controle usadas? As variáveis têm nomes significativos que refletem suas funções?
Documentação	Qual documentação de sistema está disponível? A documentação é completa, consistente e atualizada?
Dados	Existe um modelo de dados explicito para o sistema? Até que ponto os dados estão duplicados nos arquivos? Os dados usados pelo sistema estão atualizados e consistentes?
Desempenho	O desempenho do sistema é adequado? Os problemas de desempenho têm efeito significativo sobre os usuários do sistema?
Linguagem de programação	Os compiladores modernos estão disponíveis para a linguagem de programação usada para desenvolver o sistema? A linguagem de programação ainda é usada no desenvolvimento do sistema?
Gerenciamento de configuração	Todas as versões de todas as partes do sistema são gerenciadas por um sistema de gerenciamento de configurações? Existe uma descrição explícita das versões de componentes usadas no sistema atual?
Dados de teste	Existem dados de teste para o sistema? Existe um registro de testes de regressão realizados quando novas características forem acrescentadas ao sistema?
Habilidades de pessoal	Existem pessoas disponíveis que tenham as habilidades para manter a aplicação? Existe somente um número limitado de pessoas que compreendem o sistema?

hardware, isso pode exigir a substituição dos sistemas de aplicação. Se não existe orçamento disponível para a transformação de sistemas em determinado ano, a manutenção de sistemas pode prosseguir, mesmo que isso resulte em altos custos no longo prazo.

PONTOS-CHAVE

- O desenvolvimento e a evolução de software devem ser um processo único, integrado e iterativo que pode ser representado por meio de um modelo em espiral.
- As leis de Lehman, como a noção de que a mudança é contínua, descrevem uma série de percepções derivadas de estudos de longo prazo sobre a evolução de sistemas.
- Existem três tipos de manutenção de software: correção de defeitos, modificação de software para operar em um novo ambiente e implementação de novos requisitos ou de requisitos alterados.
- Para sistemas sob encomenda, os custos de manutenção de software geralmente excedem os custos de desenvolvimento.
- O processo de evolução de software é dirigido por solicitações de mudanças e inclui análise de impactos de mudanças, planejamento de releases e implementação de mudanças.
- A reengenharia de software está relacionada à reestruturação e nova documentação do software para torná-lo mais compreensível e mais fácil de mudar.
- O valor de mercado de um sistema legado e a qualidade do software da aplicação bem como seu ambiente devem ser avaliados para determinar se o sistema deve ser substituído, transformado ou mantido.

LEITURAS SUGERIDAS

Modernizing legacy systems: software technologies, engineering processes, and business practices. Este excelente livro aborda as questões de manutenção e evolução de software, bem como a migração de sistemas legados. A obra baseia-se em um grande estudo de caso da transformação de um sistema em COBOL para um sistema cliente–servidor em Java. (R. C. Seacord, et al., 2003, Addison-Wesley.)

The renaissance of legacy systems. Este livro está mais relacionado a um método para desenvolver sistemas legados. Contudo, inclui uma boa explicação geral sobre esses sistemas, estudos de casos que ilustram estruturas de sistemas legados e um capítulo sobre avaliação de sistemas. (I. Warren, 1998, Springer.)

EXERCÍCIOS

21.1 Explique por que um sistema de software usado em um ambiente real deve mudar ou tornar-se progressivamente menos útil.

21.2 Explique a base das leis de Lehman. Sob quais circunstâncias as leis podem não se aplicar?

21.3 Descreva brevemente os três tipos de manutenção de software. Por que algumas vezes é difícil distingui-los?

21.4 Sendo um gerente de projeto de software em uma empresa especializada no desenvolvimento de software para uma refinaria de petróleo em alto-mar, você recebeu a tarefa de descobrir os fatores que afetam a facilidade de manutenção dos sistemas desenvolvidos em sua empresa. Sugira como você pode definir um programa para analisar o processo de manutenção e descobrir as métricas apropriadas de facilidade de manutenção para sua empresa.

21.5 Com base na Figura 21.6, você pode observar a análise de impactos de um importante subprocesso no processo de evolução de software. Usando um diagrama, sugira quais atividades podem ser envolvidas na análise de impactos de mudanças.

21.6 Quais são os principais fatores que afetam os custos de uma reengenharia de sistemas?

21.7 Quais são as condições essenciais para que a reengenharia de software seja bem-sucedida?

21.8 Sob quais circunstâncias uma organização pode decidir descartar um sistema quando a avaliação deste sugere que ele é de alta qualidade e de alto valor de mercado?

21.9 Quais são as opções de estratégia para a evolução de um sistema legado? Quando você normalmente substituiria o todo ou parte de um sistema em vez de prosseguir com a manutenção de software (com ou sem reengenharia)?

21.10 Explique por que os problemas com apoio de software podem significar que uma organização deve substituir seus sistemas legados.

21.11 Os engenheiros de software têm responsabilidade profissional em produzir código que possa ser prontamente evoluído, mesmo que isso não seja explicitamente requisitado pelo seu empregador?

21.12 A gerência de uma organização solicitou que você realize uma avaliação de sistema e sugeriu ainda que eles gostariam de ter os resultados dessa avaliação para mostrar que o sistema é obsoleto e que deve ser substituído por um novo sistema. Isso significará que o número de profissionais envolvidos na manutenção do sistema perderão seus empregos. Sua avaliação mostra, na realidade, que o sistema está bem mantido e possui alto valor de qualidade e alto valor de mercado. Como você pode relatar esses resultados para a gerência da organização?

PARTE 5 VERIFICAÇÃO E VALIDAÇÃO

A aplicação de testes em um programa é a maneira mais comum de verificar se ele atende a sua especificação e realiza o que o cliente deseja. Entretanto, os testes são apenas uma das várias técnicas de verificação e validação. Algumas dessas técnicas, como as inspeções de programas, têm sido usadas por quase trinta anos, mas ainda não se tornaram parte da corrente principal da engenharia de software.

Nesta parte do livro, apresento abordagens para verificar se o software atende a sua especificação e para validar se ele atende às necessidades do cliente. Esta parte possui três capítulos, sendo que cada um está relacionado a diferentes aspectos de verificação e validação:

1. O Capítulo 22 apresenta uma visão geral sobre as abordagens de verificação e validação de programas. Explico a diferença entre verificação e validação e o processo de planejamento de V & V. A seguir, prossigo com a descrição de técnicas estáticas de verificação de sistema. São técnicas nas quais são verificados o código-fonte do programa sem testá-lo. Explico inspeções de programas, o uso de análise estática automatizada e, finalmente, o papel dos métodos formais no processo de verificação.

2. Testes do programa é o tópico do Capítulo 23. Explico como, normalmente, os testes são realizados em diferentes níveis e as diferenças entre teste de componente e teste de sistema. Usando exemplos simples, apresento uma série de técnicas que você pode usar para projetar casos de teste para programas e, finalmente, explico brevemente a automação de testes. A automação de testes é o uso de ferramentas de software para ajudar a reduzir o tempo e o esforço envolvidos nos processos de teste.

3. O Capítulo 24 enfoca o tópico mais especializado da validação de sistemas críticos. Em sistemas críticos, pode ser necessário provar para um cliente ou um regulamentador externo que o sistema atende a sua especificação e requisitos de confiabilidade. Descrevo abordagens para confiabilidade e para avaliação de segurança e de proteção e explico como as evidências de processos de V & V do sistema podem ser usadas no desenvolvimento de um caso de confiabilidade do sistema.

22

Verificação e validação

Objetivos

O objetivo deste capítulo é apresentar a verificação e a validação de software com foco particularmente voltado para as técnicas de verificação estática. Após ler este capítulo, você:

- compreenderá as diferenças entre verificação e validação de software;
- terá sido apresentado às inspeções de programas como um método para descobrir defeitos em programas;
- compreenderá o que é análise estática e como é usada em verificação e validação;
- compreenderá como a verificação estática é usada no processo de desenvolvimento Cleanroom.

Conteúdo

22.1 Planejamento de verificação e validação
22.2 Inspeções de software
22.3 Análise estática automatizada
22.4 Verificação e métodos formais

Durante e depois do processo de implementação, o programa em desenvolvimento deve ser verificado para certificar-se de que ele atende a sua especificação e entrega a funcionalidade esperada pelas pessoas que pagam pelo software. *Verificação e validação* (V & V) é a denominação dada a esses processos de verificação e análise. Atividades de verificação e validação ocorrem em cada estágio do processo de software. V & V começa com revisões de requisitos e continua ao longo das revisões de projeto e das inspeções de código até o teste de produto.

Verificação e validação não são a mesma coisa, embora sejam freqüentemente confundidas. Boehm (Boehm, 1979) expressou sucintamente a diferença entre elas:

- 'Validação: Estamos construindo o produto correto?'
- 'Verificação: Estamos construindo o produto corretamente?'

Essas definições nos dizem que o papel da verificação envolve verificar se o software está de acordo com suas especificações. Você deve verificar se ele atende aos requisitos funcionais e não-funcionais especificados. Validação, no entanto, é um processo mais geral. A finalidade da validação é assegurar que o sistema de software atenda às expectativas do cliente. Vai além de verificar se o sistema está conforme a sua especificação para mostrar que o software realiza o que o cliente espera que ele faça. Conforme explico na Parte 2, especificações de sistema de software nem sempre refletem os reais desejos ou necessidades de usuários e de proprietários de sistema.

O objetivo principal do processo de verificação e validação é estabelecer confiança de que o sistema de software está 'adequado a seu propósito'. Isso significa que o sistema deve ser bom o suficiente para o uso pretendido. O nível de confiabilidade exigido depende do propósito do sistema, das expectativas dos usuários do sistema e do atual ambiente de mercado:

1. *Função do software.* O nível de confiabilidade necessário depende do quão crítico é o software para uma organização. Por exemplo, o nível de confiabilidade necessário para o software usado para controlar um sistema crítico de segurança é muito mais alto do que aquele necessário para um sistema de software protótipo desenvolvido para demonstrar algumas idéias novas.
2. *Expectativas do usuário.* É um reflexo triste, na indústria de software, que muitos usuários tenham baixas expectativas de seu software e não se surpreendam quando ele falha durante o uso. Eles tendem a aceitar essas falhas de sistema quando os benefícios do uso ultrapassam as desvantagens. Entretanto, a tolerância do usuário quanto às falhas do sistema tem decrescido desde a década de 1990. Agora é menos aceitável entregar sistemas não confiáveis e, assim, empresas de software devem dedicar mais esforço para verificação e validação.
3. *Ambiente de mercado.* Quando um sistema é comercializado, os fornecedores do sistema devem levar em conta os programas dos concorrentes, o preço que os clientes estão dispostos a pagar por um sistema e o cronograma exigido para entregar esse sistema. Quando uma empresa tem poucos concorrentes, ela pode decidir liberar um programa antes que ele tenha sido inteiramente testado e depurado porque querem ser os primeiros no mercado. Quando os clientes não estão dispostos a pagar altos preços pelo software, podem estar dispostos a tolerar mais defeitos de software. Todos esses fatores devem ser considerados na decisão de quanto de esforço deve ser empregado no processo de V & V.

Dentro do processo de V & V, há duas abordagens complementares para a verificação e análise de sistema:

1. *Inspeções de software ou revisões por pares* analisam e verificam representações de sistemas como documento de requisitos, diagramas de projeto e código-fonte de programa. Você pode usar inspeções em todos os estágios do processo. Inspeções podem ser suplementadas por alguma análise automática de texto-fonte de um sistema ou de documentos associados. Inspeções de software e análises automatizadas são técnicas de V & V estáticas, quando você não precisa executar o software em um computador.
2. *Testes de software* envolvem executar uma implementação do software com dados de teste. Você examina as saídas do software e seu comportamento operacional para verificar se seu desempenho está conforme necessário. O teste é uma técnica dinâmica de verificação e validação.

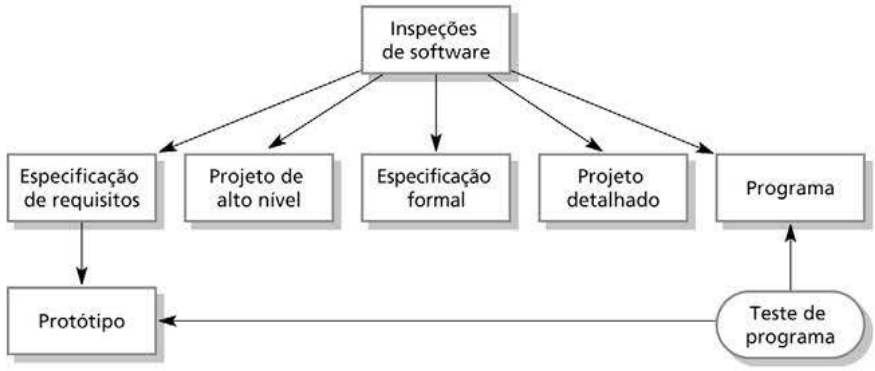
A Figura 22.1 mostra que inspeções e testes de software desempenham um papel complementar no processo de software. As setas indicam os estágios do processo em que as técnicas podem ser usadas. Portanto, você pode usar as inspeções de software em todos os estágios do processo de software. Iniciando-se com os requisitos, qualquer representação legível do software pode ser inspecionada. Conforme expliquei, revisões de requisitos e de projeto são as principais técnicas usadas para a detecção de erros na especificação e no projeto.

Você somente pode testar um sistema quando um protótipo ou uma versão executável do programa está disponível. Uma vantagem do desenvolvimento incremental é que uma versão passível de teste do sistema está disponível em estágio bem adiantado no processo de desenvolvimento. A funcionalidade pode ser testada conforme é adicionada ao sistema, dessa maneira você não precisa ter uma implementação completa antes de iniciar os testes.

Técnicas de inspeção incluem inspeções de programa, análise de código-fonte automatizada e verificação formal. Entretanto, as técnicas estáticas podem somente verificar a correspondência entre um programa e sua especificação (verificação); elas não podem demonstrar que o software é útil operacionalmente. Você também não pode usar técnicas estáticas para verificar propriedades emergentes do software como desempenho e confiabilidade.

Figura 22.1

Verificação e validação dinâmica e estática.



Embora inspeções de software sejam, atualmente, usadas amplamente, o teste de programa será sempre a técnica principal de verificação e validação de software. Testar envolve exercitar o programa usando dados como dados reais processados pelo programa. Você descobre defeitos de programa ou inadequações por meio do exame das saídas do programa e ao observar anomalias. Há dois tipos distintos de teste que podem ser usados em estágios diferentes no processo de software:

1. *Teste de validação* tem a finalidade de mostrar que o software é o que o cliente deseja — que ele atende a seus requisitos. Como parte do teste de validação, você pode usar testes estatísticos para testar o desempenho e a confiabilidade do programa e para verificar como ele funciona sob condições operacionais. Explico testes estatísticos e estimativa de confiabilidade no Capítulo 24.
2. *Teste de defeitos* é destinado a revelar defeitos no sistema em vez de simular o seu uso operacional. O objetivo do teste de defeitos é encontrar inconsistências entre um programa e sua especificação. Abordo teste de defeitos no Capítulo 23.

Evidentemente, não há uma delimitação rígida e rápida entre essas abordagens para teste. Durante o teste de validação, você encontrará defeitos no sistema; durante o teste de defeitos, alguns dos testes mostrarão que o programa atende a seus requisitos.

Os processos de V &V e de debugging normalmente são intercalados. À medida que descobre defeitos no programa que está sendo testado, você deve alterar o programa para corrigir esses defeitos. Entretanto, teste (ou, mais genericamente, verificação e validação) e debugging têm finalidades diferentes:

1. Processos de verificação e validação são dedicados a estabelecer a existência de defeitos em um sistema de software.
2. Debugging é um processo (Figura 22.2) que localiza e corrige esses defeitos.

Não há método simples para debugging de programas. Pessoas experientes que realizam debugging procuram por padrões nas saídas do teste nas quais o defeito se manifesta e usam os conhecimentos sobre o tipo de defeito, o padrão de saída, a linguagem de programação e o processo de programação para localizar o defeito. Ao realizar o debugging, você pode usar seus próprios conhecimentos a respeito dos erros comuns de programadores (como deixar de incrementar um contador) e confrontá-los com os padrões observados. Você deve também procurar por erros típicos característicos da linguagem de programação, como erro de direcionamento de ponteiro em C.

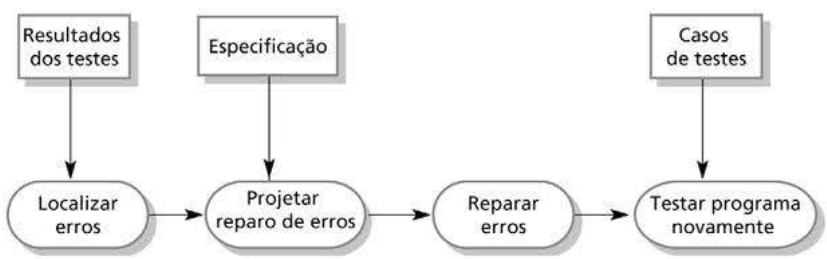
Localizar os defeitos em um programa não é sempre um processo simples, já que o defeito pode não estar perto do ponto em que o programa falhou. Para localizar um defeito de programa, você pode ter de projetar testes adicionais que reproduzam o defeito original e que apontem sua localização no programa. Você pode ter de rastrear o programa manualmente, linha por linha. As ferramentas de debugging que coletam informações sobre a execução do programa também podem ajudar a localizar a origem de um problema.

As ferramentas de debugging interativas são, em geral, parte de um conjunto de ferramentas de apoio da linguagem que estão integradas com um sistema de compilação. Elas fornecem um ambiente em run-time especializado para o programa que permite acessar a tabela de símbolos do compilador e, a partir daí, acessar os valores das variáveis do programa. Você pode controlar a execução 'passo a passo' do programa. Após cada declaração ser executada, podem-se examinar os valores das variáveis e assim descobrir a localização do defeito.

Depois que um defeito do programa foi descoberto, você precisa corrigi-lo e revalidar o sistema. Isso pode envolver uma nova inspeção do programa ou teste de regressão, nos quais testes existentes são executados novamente. O teste de regressão é usado para verificar se as mudanças feitas em um programa não introduziram novos defeitos. A experiência tem mostrado que uma alta proporção de 'reparos' de defeitos é incompleta ou introduz novos defeitos no programa.

Em princípio, você deve repetir todos os testes após cada reparo de defeitos; na prática, isso é muito dispendioso. Como parte do plano de teste, você deve identificar dependências entre componentes e os testes associados com cada componente. Ou seja, deve haver rastreabilidade dos casos de testes aos componentes testados. Se essa rastreabilidade for documentada, você poderá executar um subconjunto de casos de testes de sistema a fim de verificar o componente modificado e seus dependentes.

Figura 22.2
Processo de debugging.



22.1 Planejamento de verificação e validação

Verificação e validação é um processo dispendioso. Para alguns sistemas, como os sistemas de tempo real com restrições não-funcionais complexas, mais da metade do orçamento de desenvolvimento do sistema pode ser gasto em V & V. Um planejamento cuidadoso é necessário para se obter o máximo de inspeções e testes e para controlar os custos do processo de verificação e validação.

Você deve iniciar antecipadamente o planejamento de verificação e validação de sistema no processo de desenvolvimento. O modelo de processo de desenvolvimento de software mostrado na Figura 22.3 é chamado, algumas vezes, de modelo V (coloque a Figura 22.3 na vertical e perceba o V). É uma instância do modelo cascata genérico (veja o Capítulo 4) e mostra que os planos de testes devem ser derivados da especificação e do projeto de sistema. Esse modelo também divide o sistema V & V em vários estágios. Cada estágio é dirigido por testes definidos a fim de verificar a conformidade do programa com seu projeto e sua especificação.

Como parte do processo de planejamento de V & V, você deve decidir sobre o equilíbrio entre as abordagens estática e dinâmica para verificação e validação, especificar padrões e procedimentos para inspeções e testes de software, estabelecer checklists para orientar as inspeções de programa (veja a Seção 22.3) e definir o plano de teste de software.

O esforço relativo dedicado às inspeções e aos testes depende do tipo de sistema que está sendo desenvolvido e da habilidade organizacional com a inspeção de programas. Como regra geral, quanto mais crítico o sistema, mais esforço deve ser dedicado às técnicas de verificação estática.

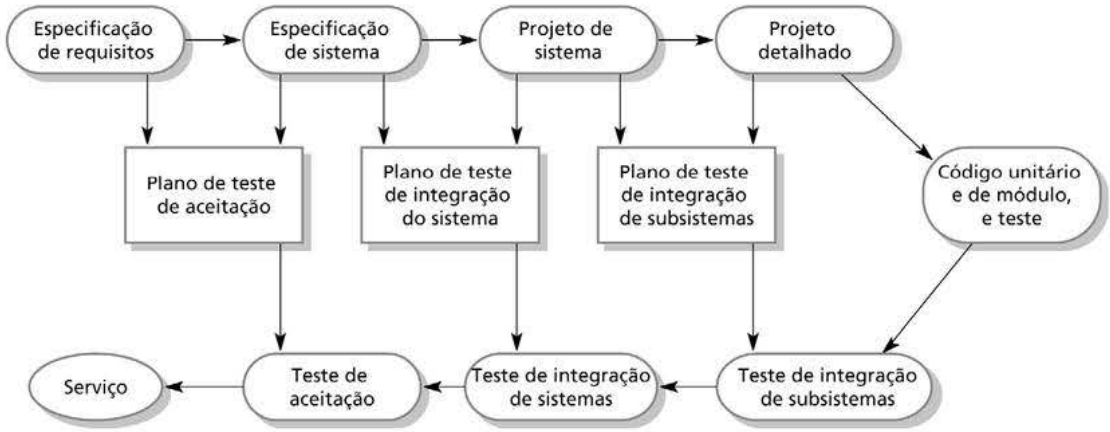
O planejamento de teste se concentra em estabelecer padrões para o processo de teste, não apenas em descrever testes de produtos. Assim como ajudam os gerentes a alocar recursos e estimar cronogramas de testes, os planos de teste são destinados aos engenheiros de software envolvidos a projetar e realizar os testes de sistemas. Eles ajudam o pessoal técnico a obter um quadro geral dos testes de sistema e situar seu próprio trabalho no contexto. Uma boa descrição de planos de teste e sua relação com planos de qualidade mais gerais é dada em Frewin e Hatton (Frewin e Hatton, 1986). Humphrey (Humphrey, 1989) e Kit (Kit, 1995) também incluem explicações sobre planejamento de teste.

Os componentes principais de um plano de teste para um sistema grande e complexo são mostrados no Quadro 22.1. Assim como estabelece o cronograma e os procedimentos de teste, o plano de teste define os recursos de hardware e de software necessários. Isso é útil para gerentes de sistema responsáveis por garantir que esses recursos estarão disponíveis para a equipe de testes. Planos de teste devem, normalmente, incluir quantidades significativas de contingência de modo que enganos no projeto e na implementação possam ser acomodados e o pessoal possa ser realocado a outras atividades.

Para sistemas menores, um plano de testes menos formal pode ser usado, mas ainda há uma necessidade de um documento formal para apoiar o planejamento do processo de testes. Para alguns processos ágeis, como *extreme programming*, o teste é inseparável do desenvolvimento. Como outras atividades de planejamento, o planejamento de teste também é incremental. No XP, o cliente é fundamentalmente responsável pela decisão do quanto de esforço deve ser dedicado ao teste de sistema.

Planos de teste não são documentos estáticos; evoluem durante o processo de desenvolvimento. Os planos de teste mudam devido a atrasos em outros estágios do processo de desenvolvimento. Se parte de um sistema está incompleta, o sistema como um todo não pode ser testado. Você tem, portanto, de atualizar o plano de teste para realocar os testadores para alguma outra atividade e trazê-los de volta quando o software estiver disponível novamente.

Figura 22.3 Plano de teste como ligação entre o desenvolvimento e os testes.



Quadro 22.1

Estrutura de um plano de teste de software.

Processo de teste

Descrição das fases principais do processo de teste. Essas fases podem ser como as descritas anteriormente neste capítulo.

Rastreabilidade de requisitos

Os usuários são os mais interessados em que o sistema atenda a seus requisitos e em que os testes sejam planejados de maneira que todos os requisitos sejam individualmente testados.

Itens testados

Os produtos do processo de software a serem testados devem ser especificados.

Cronograma de testes

Um cronograma global de testes e alocação de recursos para esse cronograma está, obviamente, vinculado ao cronograma mais geral de desenvolvimento de projeto.

Procedimentos de registro de testes

Não é suficiente realizar simplesmente os testes; os resultados dos testes devem ser sistematicamente registrados. Deve ser possível auditar o processo de teste para verificar que foi conduzido corretamente.

Requisitos de hardware e de software

Esta seção deve estabelecer as ferramentas de software necessárias e a utilização estimada de hardware.

Restrições

As restrições que afetam o processo de teste, como falta de pessoal, devem ser antecipadas nesta seção.

22.2 Inspeções de software

Inspeções de software é processo de V & V estático, no qual um sistema de software é revisto para se encontrar erros, omissões e anomalias. Geralmente, as inspeções enfocam o código-fonte, mas qualquer representação legível do software, como seus requisitos ou um modelo de projeto, pode ser inspecionado. Quando se inspeciona um sistema, você usa o conhecimento do sistema, seu domínio de aplicação e a linguagem de programação ou o modelo de projeto para se descobrir erros.

Há três vantagens principais da inspeção sobre o teste:

1. Durante os testes, erros podem mascarar (ocultar) outros erros. Uma vez que um erro é descoberto, você nunca pode estar seguro de se outras anomalias de saída são devidas a um novo erro ou se são efeitos colaterais do erro original. Pelo fato de a inspeção ser um processo estático, você não precisa se preocupar com interações entre os erros. Conseqüentemente, uma única sessão de inspeção pode descobrir muitos erros de um sistema.
2. Versões incompletas de um sistema podem ser inspecionadas sem custos adicionais. Se um programa está incompleto, você precisa desenvolver conjuntos de testes especializados para as partes disponíveis. Isso obviamente acrescenta custos adicionais ao sistema.
3. Assim como procurar por defeitos de programas, uma inspeção pode também considerar atributos de qualidade mais amplos de um programa como conformidade com padrões, portabilidade e facilidade de manutenção. Você pode procurar por ineficiências, algoritmos inapropriados e estilo de programação "pobre" que poderiam tornar difíceis a manutenção e a atualização de sistema.

Inspeções são uma idéia antiga. Tem havido vários estudos e experimentos que têm demonstrado que as inspeções são mais eficientes para descobrir defeitos do que teste de programas. Fagan (Fagan, 1986) relatou que mais de 60% dos erros em um programa podem ser detectados usando-se inspeções informais de programa. Mill et al. (Mills, et al., 1987) sugerem que uma abordagem mais formal para inspeção baseada em argumentos de correção pode detectar mais de 90% dos erros de um programa. Essa técnica é usada no processo Cleanroom descrito na Seção 22.4. Selby e Basili (Selby, et al., 1987) compararam, empiricamente, a eficiência das inspeções e dos testes. Eles constataram que a revisão estática de código era mais eficiente e menos dispendiosa do que teste de defeitos no descobrimento de defeitos de programas. Gilb e Graham (Gilb e Graham, 1993) constataram também que isso era verdadeiro.

Revisões e testes têm, cada um, vantagens e desvantagens que devem ser usadas em conjunto no processo de verificação e validação. Realmente, Gilb and Graham sugerem que um dos usos mais eficientes das revisões é revisar os casos de testes para um sistema. As revisões podem descobrir problemas com esses testes e ajudar a projetar caminhos mais eficientes para testar o sistema. Você pode começar o V & V do sistema com inspeções no início do processo de desenvolvimento, mas, uma vez que um sistema esteja integrado, é preciso testá-lo para verificar as propriedades emergentes e se a funcionalidade do sistema é a que seu proprietário realmente deseja.

Apesar do sucesso das inspeções, foi provado que é difícil introduzir inspeções formais dentro de muitas organizações de desenvolvimento de software. Engenheiros de software com experiência em teste de programa, algumas vezes, relutam em aceitar que as inspeções podem ser mais eficientes para detectar defeitos do que os testes. Gerentes podem ficar intrigados a respeito do motivo pela qual as inspeções exigem custos adicionais durante o projeto e o desenvolvimento. Eles podem não desejar assumir o risco de que não haverá economias correspondentes durante o teste de programa.

Não há dúvida de que as inspeções incorrem no início os custos de V & V do software e resultam em economias de custo somente depois que as equipes de desenvolvimento se tornam experientes em seu uso. Além disso, há os problemas práticos relacionados à organização de inspeções: inspeções levam tempo para serem organizadas e parecem diminuir a velocidade do processo de desenvolvimento. É difícil convencer um gerente extremamente pressionado de que esse tempo pode ser recuperado mais tarde pelo fato de que menos tempo será gasto no debugging do programa.

22.2.1 Processo de inspeção de programa

Inspeções de programa são revisões cujo objetivo é a detecção de defeitos de programa. A noção de um processo de inspeção formalizado foi desenvolvida pela primeira vez na IBM, na década de 1970 (Fagan, 1976; Fagan,1986). Atualmente, a inspeção é um método de verificação de programa amplamente usado, especialmente em engenharia de sistemas críticos. Do método original de Fagan, uma série de abordagens alternativas para inspeção foi desenvolvida (Gilb e Graham, 1993). Elas são todas baseadas numa equipe com membros de diferentes experiências que faz uma revisão cuidadosa, linha por linha, de código-fonte do programa.

A diferença fundamental entre inspeções de programa e outros tipos de revisão de qualidade é que o objetivo específico das inspeções é encontrar os defeitos de programa, mais do que considerar questões mais amplas de projeto. Os defeitos podem ser erros de lógica, anomalias no código que possam indicar uma condição errônea ou não-conformidade aos padrões do projeto ou organizacionais. Ao contrário, outros tipos de revisão podem ser mais relacionados com cronograma, custos, progresso em comparação aos marcos de projeto ou avaliação de se o software vai atender aos objetivos organizacionais.

A inspeção de programa é um processo formal realizado por uma equipe de pelo menos quatro pessoas. Os membros da equipe analisam sistematicamente o código e apontam possíveis defeitos. Nas propostas originais de Fagan, ele sugeriu papéis como autor, leitor, testador e moderador. O leitor lê o código em voz alta para a equipe de inspeção, o testador inspeciona o código sob a perspectiva de teste e o moderador organiza o processo.

À medida que as organizações ganharam experiência em inspeção, emergiram outras propostas para funções da equipe. Em uma explicação de como a inspeção foi bem-sucedida no processo de desenvolvimento da Hewlett-Packard, Grady e Van Slack (Grady e Van Slack, 1994) sugerem seis papéis, conforme mostrados na Tabela 22.1. Eles não acham que seja necessária a leitura do programa em voz alta. A mesma pessoa pode desempenhar mais de um papel e, assim, o tamanho da equipe pode variar de uma inspeção para outra. Gilb e Graham sugerem que os inspetores devem ser selecionados para refletir diferentes pontos de vista, como teste, usuário final e gerenciamento de qualidade.

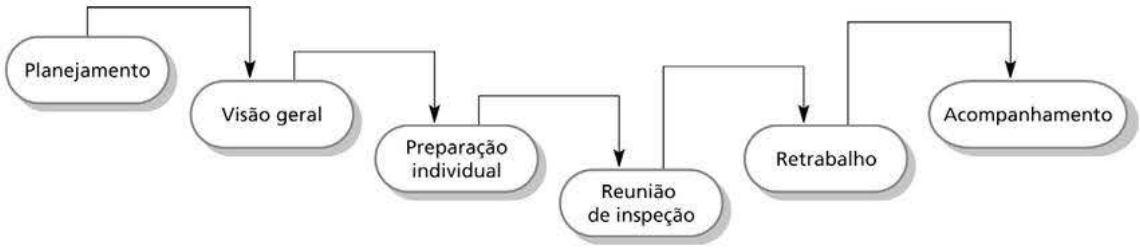
As atividades em um processo de inspeção são mostradas na Figura 22.4. Antes do início de um processo de inspeção de programa, é essencial que:

1. Você tenha uma especificação precisa do código a ser inspecionado. É impossível inspecionar um componente no nível de detalhes necessário para detectar defeitos sem uma especificação completa.

Tabela 22.1 Papéis no processo de inspeção

Papel	Descrição
Autor e proprietário	O programador ou projetista responsável por produzir o programa ou o documento. Ele é responsável pela correção de defeitos descobertos durante o processo de inspeção.
Inspetor	Encontra erros, omissões e inconsistências nos programas e documentos. Pode também identificar questões mais amplas fora do escopo da equipe de inspeção.
Leitor	Apresenta o código ou documento em uma reunião de inspeção.
Relator	Registra os resultados da reunião de inspeção.
Presidente ou moderador	Gerencia o processo e facilita a inspeção. Relata os resultados do processo ao moderador-chefe.
Moderador-chefe	Responsável pelos aprimoramentos do processo de inspeção, pela atualização da lista de verificação, pelo desenvolvimento de padrões etc.

Figura 22.4 Processo de inspeção.



2. Os membros da equipe de inspeção estejam familiarizados com os padrões organizacionais.
3. Uma versão atualizada e compilável do código tenha sido distribuída para todos os membros da equipe. Não há razão para se inspecionar um código que seja 'quase completo', mesmo que um atraso cause uma interrupção de cronograma.

O moderador da equipe de inspeção é responsável pelo planejamento da inspeção. Isso envolve selecionar uma equipe de inspeção, organizar uma sala de reunião e assegurar que o material a ser inspecionado e suas especificações estejam completos. O programa a ser inspecionado é apresentado para a equipe de inspeção durante o estágio de visão geral quando o autor do código descreve o que o programa se destina a fazer. Isso é seguido por um período de preparação individual. Cada membro da equipe de inspeção estuda a especificação e o programa, e procura defeitos no código.

A inspeção propriamente dita deve ser bem curta (não deve durar mais do que duas horas) e deve enfocar a detecção de defeitos, conformidade aos padrões e programação de baixa qualidade. A equipe de inspeção não deve sugerir como esses defeitos devem ser corrigidos nem recomendar mudanças em outros componentes.

Continuando a inspeção, o autor do programa deve fazer mudanças nele para corrigir os problemas identificados. No estágio seguinte, o moderador deve decidir se uma nova inspeção do código é necessária. Ele pode decidir que uma nova inspeção completa não é necessária e que os defeitos foram corrigidos com sucesso. Então, o programa é aprovado pelo moderador para ser liberado.

Durante uma inspeção, um checklist de erros de programação comuns é freqüentemente usada para enfocar a discussão. Esse checklist pode ser baseado em exemplos de checklists de livros ou de conhecimento de defeitos comuns em determinado domínio de aplicação. Você precisa de checklists diferentes para linguagens de programação diferentes, porque cada linguagem tem seus próprios erros característicos. Humphrey (Humphrey, 1989), em ampla explicação sobre inspeções, fornece uma série de exemplos de checklists de inspeção.

Esse checklist varia de acordo com a linguagem de programação por causa dos diferentes níveis de verificação providos por um compilador de linguagem. Por exemplo, um compilador Java verifica se as funções têm o número correto de parâmetros; um compilador C não faz isso. Verificações possíveis que poderiam ser feitas durante o processo de inspeção são mostradas na Tabela 22.2. Gilb e Graham (Gilb e Graham, 1993) enfatizam que cada organização deve desenvolver seu próprio checklist de inspeção baseado nos padrões locais e na prática. Os checklists devem ser atualizados regularmente conforme novos tipos de defeitos são encontrados.

O tempo necessário para uma inspeção e a quantidade de código que pode ser coberto depende da experiência da equipe de inspeção, da linguagem de programação e do domínio de aplicação. Ambos, Fagan da IBM e Barnard e Price (Barnard e Price, 1994), que avaliaram o processo de inspeção para software de telecomunicações, chegaram a conclusões semelhantes:

1. Cerca de 500 declarações de código-fonte por hora podem ser apresentadas durante o estágio de visão geral.
2. Durante a preparação individual, cerca de 125 declarações de código-fonte por hora podem ser examinadas.
3. De 90 a 125 declarações por hora podem ser inspecionadas durante a reunião de inspeção.

Com quatro pessoas envolvidas em uma equipe de inspeção, o custo de inspeção de 100 linhas de código é aproximadamente equivalente ao esforço de uma pessoa-dia. Isso mostra que a inspeção por si consome cerca de uma hora e que cada membro da equipe gasta de uma a duas horas preparando-se para a inspeção. Os custos de teste variam muito e depende do número de defeitos no programa. Contudo, o esforço exigido pela inspeção do programa é provavelmente menos da metade do esforço que seria necessário para os testes de defeitos equivalentes.

Algumas organizações (Gilb e Graham, 1993) abandonaram atualmente o teste de componente em favor de inspeções. Eles constataram que as inspeções de programa são mais eficientes para encontrar erros, e que os custos de teste de componente não são justificáveis. Essas organizações constataram que as inspeções de componentes, combinadas com teste de sistema, eram a estratégia de V & V de custo mais adequado. Conforme explico mais adiante no capítulo, essa abordagem é usada no processo de desenvolvimento de software Cleanroom.

Tabela 22.2 Verificações de inspeção

Classe de defeitos	Verificação de inspeção
Defeitos de dados	Todas as variáveis de programa são iniciadas antes que seus valores sejam usados? Todas as constantes foram denominadas? O limite superior de vetores deve ser igual ao tamanho do vetor ou Tamanho –1? Se são usados strings de caracteres, um delimitador é explicitamente atribuído? Existe alguma possibilidade de overflow de buffer?
Defeitos de controle	Para cada declaração condicional, a condição está correta? Cada loop está terminando corretamente? As declarações compostas estão corretamente delimitadas entre parênteses? Em declarações 'case', todos os casos possíveis são levados em conta? Se um comando 'break' é necessário após cada caso nas declarações 'case', ele foi incluído?
Defeitos de entrada/saída	Todas as variáveis de entrada são usadas? Todas as variáveis de saída têm valor atribuído antes de sua saída? Entradas inesperadas podem fazer com que os dados sejam corrompidos?
Defeitos de interface	Todas as chamadas de funções e de métodos têm o número correto de parâmetros? Tipos de parâmetros reais e formais se combinam? Os parâmetros estão na ordem correta? Se os componentes acessam memória compartilhada, eles têm o mesmo modelo de estrutura de memória compartilhada?
Defeitos de gerenciamento de armazenamento	Se uma estrutura ligada é modificada, todas as ligações foram corretamente reatribuídas? Se o armazenamento dinâmico foi usado, o espaço foi corretamente alocado? O espaço de memória é liberado depois de não ser mais necessário?
Defeitos de gerenciamento de exceções	Todas as condições possíveis de erro foram consideradas?

A introdução de inspeções tem implicações para o gerenciamento do projeto. Gerenciamento cauteloso é importante caso as inspeções devam ser aceitas pelas equipes de desenvolvimento de software. Inspeção de programa é um processo público de detecção de erros comparado com o processo de teste de componente mais privativo.

Inevitavelmente, erros cometidos por indivíduos são revelados para toda a equipe de programação. Os líderes da equipe de inspeção devem ser treinados para gerenciar o processo cuidadosamente e para desenvolver uma cultura que forneça apoio sem recriminação quando erros são descobertos.

À medida que uma organização ganha experiência no processo de inspeção, ela pode usar os resultados de uma inspeção para ajudar no aprimoramento do processo. As inspeções são um modo ideal de coletar dados de tipos de defeitos que ocorrem. A equipe de inspeção e os autores do código inspecionado podem sugerir as razões pelas quais esses defeitos foram introduzidos. Sempre que possível, o processo deve ser modificado para eliminar as razões para os defeitos, tornando, assim, possível que eles sejam evitados nos sistemas futuros.

22.3 Análise estática automatizada

Inspeções são uma forma de análise estática — você examina o programa sem executá-lo. Conforme expliquei, as inspeções são freqüentemente dirigidas por checklists de erros e heurísticas que identificam erros comuns em diferentes linguagens de programação. Para alguns erros e heurísticas, é possível automatizar o processo de verificação de programas em relação a essa lista, o que resultou no desenvolvimento de analisadores estáticos automatizados para diferentes linguagens de programação.

Analisadores estáticos são ferramentas de software que varrem o texto-fonte de um programa e detectam possíveis defeitos e anomalias. Eles analisam o texto de programa e, assim, reconhecem os tipos de declarações no programa. Podem portanto detectar se as declarações estão bem formuladas, fazer inferências sobre o fluxo de controle do programa e, em muitos casos, computam o conjunto de todos os valores possíveis para os dados de programa. Eles complementam os recursos de detecção de erros providos pelo compilador da linguagem. Podem ser usados como parte do processo de inspeção ou como uma atividade separada do processo de V & V.

A intenção da análise estática automática é chamar a atenção do inspetor para anomalias do programa, como variáveis usadas sem serem iniciadas, variáveis não usadas ou dados cujos valores poderiam ficar fora de sua extensão. Algumas das verificações que podem ser detectadas por meio de análise estática são mostradas na Tabela 22.3. As anomalias são freqüentemente um resultado de erros de programação ou omissões, de modo que eles enfatizam coisas que poderiam sair erradas quando o programa fosse executado. Entretanto, você deve compreender que essas anomalias não são necessariamente defeitos de programa. Eles podem ser determinados ou não ter conseqüências adversas.

Os estágios envolvidos na análise estática incluem:

1. *Análise de fluxo de controle.* Esse estágio identifica e enfatiza os loops com vários pontos de saída ou de entrada e código inacessível. Código inacessível é o código cercado por declarações incondicionais 'goto' ou em uma ramificação de uma declaração condicional na qual a condição de guarda nunca pode ser verdadeira.
2. *Análise de uso de dados.* Esse estágio enfatiza como as variáveis do programa são usadas. Ela detecta variáveis usadas sem prévia iniciação, variáveis escritas duas vezes sem uma tarefa de impedimento e variáveis declaradas mas que nunca são usadas. A análise de uso de dados também descobre testes não eficientes nos quais a condição do teste é redundante. Condições redundantes são condições sempre verdadeiras ou sempre falsas.
3. *Análise de interface.* Essa análise verifica a consistência das declarações de rotina e de procedimento e seus usos. Ela é desnecessária se uma linguagem com tipagem forte como Java for usada para implementação, uma vez que o compilador realiza essas verficações. A análise de interface pode detectar erros de tipo em linguagens com tipagem fraca, como Fortran e C. A análise de interface pode também detectar funções e procedimentos declarados e nunca chamados ou resultados de funções que nunca são usados.
4. *Análise de fluxo de informações.* Esse estágio de análise identifica as dependências entre variáveis de entrada e de saída. Embora ele não detecte anomalias, mostra como o valor de cada variável de programa é derivada de outros valores de variáveis. Com essas informações, a inspeção de código deve ser capaz de encontrar valores que foram erroneamente computados. A análise de fluxo de informações pode mostrar também as condições que afetam um valor de variável.
5. *Análise de caminho.* Esse estágio de análise semântica identifica todos os caminhos possíveis por meio do programa e estabelece as declarações executadas naquele caminho. Essencialmente, ela elucida o controle do programa e permite que cada predicado possível seja analisado individualmente.

Analisadores estáticos são particularmente valiosos quando uma linguagem de programação, como C, for usada. A linguagem C não possui regras de tipagem estritas e a verificação que o compilador C pode executar é limitada. Portanto, é fácil para programadores cometerem erros e a ferramenta de análise estática pode descobrir automaticamente alguns dos defeitos de programa resultantes. Isso é particularmente importante quando C (e, em menor extensão, C++) for usada para o desenvolvimento de sistemas críticos. Nesse caso, a análise estática pode descobrir um grande número de erros potenciais e reduzir significativamente os custos de teste.

Não há dúvida de que, para uma linguagem como C, a análise estática é uma técnica eficiente para descobrir erros de programa. Isso compensa pelas fraquezas no projeto de linguagem de programação. Entretanto, os projetistas das lingua-

Tabela 22.3 Verificação de análise estática automatizada

Classe de defeitos	Verificação de análise estática
Defeitos de dados	Variáveis usadas antes da inicialização Variáveis declaradas, mas nunca usadas Variáveis atribuídas duas vezes, mas nunca usadas entre atribuições Possíveis violações de limites de vetor Variáveis não declaradas
Defeitos de controle	Código inacessível Ramificações incondicionais em loops
Defeitos de entrada/saída	Variáveis geradas duas vezes sem tarefa de impedimento
Defeitos de interface	Tipo de parâmetro que não combina Número de parâmetro que não combina Resultados de funções não usadas Funções e procedimentos não chamados
Defeitos de gerenciamento de armazenamento	Ponteiros não atribuídos Aritmética de ponteiros

Figura 22.5

Análise estática com o Lint.

```
138% more lint_ex.c
#include <stdio.h>
printarray (Anarray)
int Anarray;
{
printf("%d",Anarray);
}
main ()
{
int Anarray[5]; int i; char c;
printarray (Anarray, i, c);
printarray (Anarray) ;
}

139% cc lint_ex.c
140% lint lint_ex.c

lint_ex.c(10): warning: c may be used before set
int_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

gens de programação modernas, como Java, têm removido algumas características de linguagem propensas a erro. Todas as variáveis devem ser iniciadas, não há declarações 'goto'; assim, os códigos inacessíveis são menos prováveis de serem criados acidentalmente e o gerenciamento de armazenamento é automático. Essa abordagem de prevenção de erros em vez de detecção de erros é mais eficiente no aprimoramento da confiabilidade de programa.

Embora analisadores estáticos para Java estejam disponíveis, eles não são amplamente usados. Não está claro se o número de erros detectados justifica o tempo necessário para analisar as saídas.

Portanto, para ilustrar a análise estática, uso um programa pequeno em C em vez de um programa em Java. Os sistemas Unix e Linux incluem um analisador estático chamado Lint para programas em C. O Lint fornece verificação estática equivalente àquelas fornecidas pelo compilador em uma linguagem de tipagem forte como Java. Um exemplo de saída produzida pelo Lint é mostrado na Figura 22.5. O primeiro comando (linha 138) lista o programa (sem propósito). Ele define uma função com um parâmetro, denominado **printarray**, e, então, chama essa função com três parâmetros. As variáveis **i** e **c** são declaradas, mas nunca lhes são atribuídos valores. O valor retornado pela função nunca é usado.

A linha numerada 139 mostra a compilação C desse programa sem erros reportados pelo compilador C. Isso é seguido por uma chamada do analisador estático Lint que detecta e reporta os erros de programa.

O analisador estático mostra que as variáveis **c** e **i** foram usadas, mas não iniciadas, e que **printarray** foi chamado com um número diferente de argumentos do que foi declarado. Ele também identifica o uso inconsistente do primeiro argumento no **printarray** e o fato de que o valor de função nunca é usado.

A análise baseada em ferramentas não pode substituir as inspeções, como há alguns tipos de erros que analisadores estáticos não podem detectar. Por exemplo, eles podem detectar variáveis não iniciadas, mas não podem detectar iniciações que estão incorretas. Em linguagens de tipagem fraca, como C, os analisadores estáticos podem detectar funções que tenham números e tipos de argumentos errados, mas não podem detectar situações nas quais o argumento incorreto do tipo correto foi passado para uma função.

Para resolver alguns desses problemas, analisadores estáticos como LCLint (Orcero, 2000; Evans e Larochelle, 2002) apóiam o uso de anotações nas quais os usuários definem restrições como comentários estilizados no programa. Essas restrições permitem que um programador especifique que as variáveis em uma função não devem ser alteradas, as variáveis globais usadas etc. O analisador estático pode, depois, verificar o programa em relação a essas restrições e enfatizar as seções de código que parecem estar incorretas.

22.4 Verificação e métodos formais

Métodos formais de desenvolvimento de software são baseados em representações matemáticas de software, usualmente como uma especificação formal. Esses métodos formais estão principalmente relacionados a uma análise matemática da especificação; com uma transformação da especificação para uma representação mais detalhada e semanticamente

equivalente ou a uma verificação formal de que uma representação do sistema é semanticamente equivalente a uma outra representação.

Você pode pensar no uso de métodos formais como a última técnica de verificação estática. Eles exigem análises muito detalhadas da especificação de um sistema e de um programa e seu uso freqüentemente consome tempo e é oneroso. Em decorrência, o uso de métodos formais está, na maioria das vezes, restrito aos processos de desenvolvimento de software com segurança e proteção críticas. O uso de especificação matemática formal e verificação associada foram obrigatórios nos padrões de defesa do Reino Unido (UK) para softwares de segurança crítica. (MOD, 1995).

Métodos formais podem ser usados em diferentes estágios no processo de V & V:

1. Uma especificação formal de sistema pode ser desenvolvida e analisada matematicamente para verificar inconsistências. Essa técnica é eficiente para descobrir erros de especificação e omissões, conforme explicado no Capítulo 10.
2. Você pode verificar formalmente, por meio de argumentos matemáticos, se o código de um sistema de software está consistente com sua especificação. Isso requer uma especificação formal e é eficiente para descobrir erros de programação e de projeto. Um processo de desenvolvimento de transformação, no qual uma especificação formal é transformada por meio de uma série de representações mais detalhadas ou um processo Cleanroom pode ser usado para apoiar o processo de verificação formal.

O argumento a favor do uso de especificação formal e verificação de programa associada é que a especificação formal força uma análise detalhada da especificação. Isso pode revelar inconsistências potenciais ou omissões potenciais que não poderiam ser descobertas de outra maneira até o sistema entrar em operação. A verificação formal demonstra que o programa desenvolvido atende à sua especificação e, assim, os erros de implementação não comprometem a confiabilidade do sistema.

O argumento contra o uso de especificação formal é que ele exige notações especializadas. Essas notações podem ser usadas somente por pessoal especialmente treinado e não podem ser compreendidas por especialistas de domínio de problema. Dessa maneira, problemas com requisitos de sistema podem ser conciliados pela formalidade. Engenheiros de software não podem reconhecer dificuldades potenciais com os requisitos porque não compreendem o domínio; especialistas de domínio não podem encontrar esses problemas porque não compreendem a especificação. Embora a especificação possa ser matematicamente consistente, ela não pode especificar as propriedades de sistema realmente necessárias.

A verificação de um sistema de software não trivial demanda grande quantidade de tempo e requer ferramentas especializadas, como provadores de teorema e habilidades matemáticas. É, portanto, um processo extremamente dispendioso e, à medida que o tamanho do sistema aumenta, os custos de verificação formal aumentam desproporcionalmente. Muitas pessoas, portanto, constatam que a verificação formal não apresenta custos adequados. O mesmo nível de confiabilidade no sistema pode ser alcançado de maneira menos dispendiosa por meio do uso de outras técnicas de validação, como inspeções e teste de sistema.

Algumas vezes se afirma que o uso de métodos formais para desenvolvimento de sistema conduz a sistemas mais confiáveis e mais seguros. Não há dúvida de que uma especificação formal de sistema é menos propícia a conter anomalias que devem ser resolvidas pelo projetista de sistema. No entanto, a especificação e a prova formal não garantem que o software será confiável no uso prático. As razões para isso são:

1. *A especificação pode não refletir os requisitos reais dos usuários do sistema.* Lutz (Lutz, 1993) descobriu que muitas falhas experimentadas por usuários eram uma conseqüência de erros de especificação e omissões que não podiam ser detectados pela especificação formal de sistema. Além disso, os usuários de sistema raramente compreendem notações formais e, assim, não podem ler a especificação formal diretamente para encontrar erros e omissões.
2. *A prova pode conter erros.* As provas de programa são amplas e complexas; portanto, como programas amplos e complexos, geralmente contêm erros.
3. *A prova pode assumir um padrão de uso incorreto.* Se o sistema não é usado conforme o previsto, a prova pode ser inválida.

Apesar de suas desvantagens, minha visão (explicada no Capítulo 10) é de os que métodos formais têm um papel importante a desempenhar no desenvolvimento de sistemas de software críticos. Especificações formais são muito eficientes em descobrir problemas de especificação, que são as causas mais comuns de falha de sistema. A verificação formal aumenta a confiabiliade da maioria dos componentes críticos desses sistemas. O uso de abordagens formais está aumentando de acordo com a demanda dos clientes e à medida que cada vez mais engenheiros familiarizam-se com essas técnicas.

22.4.1 Desenvolvimento de software Cleanroom

Métodos formais foram integrados com uma série de processos de desenvolvimento de software. No método B (Wordsworth, 1996), uma especificação formal é transformada por meio de uma série de transformações de preservação

de correções no programa. SDL (Mitschele-Thiel, 2001) é usada no desenvolvimento de sistemas de telecomunicações e VDM (Jones, 1986) e Z (Spivey, 1992) foram usados nos processos do tipo cascata. Outra abordagem bem documentada que usa métodos formais é o processo de desenvolvimento Cleanroom. O desenvolvimento de software Cleanroom (Mills, et al., 1987; Cobb e Mills, 1990; Linger, 1994; Prowell, et al., 1999) é uma filosofia de desenvolvimento de software que usa métodos formais para apoiar inspeção rigorosa de software.

Um modelo de processo Cleanroom é mostrado na Figura 22.6. O objetivo dessa abordagem para o desenvolvimento de software é o software com defeito zero. O nome 'Cleanroom' foi derivado por analogia com unidades de fabricação de semicondutores, em que os defeitos são evitados por meio da manufatura em uma atmosfera ultralimpa. O desenvolvimento Cleanroom é particularmente relevante para este capítulo porque substituiu o teste unitário dos componentes de sistema por meio de inspeções para verificar a consistência desses componentes com suas especificações.

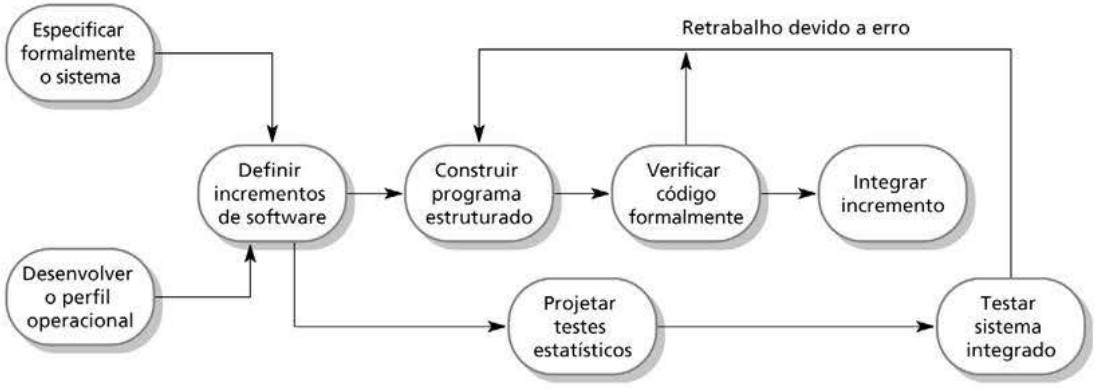
A abordagem Cleanroom para desenvolvimento de software baseia-se em cinco estratégias principais:

1. *Especificação formal*. O software a ser desenvolvido é especificado formalmente. Um modelo de estado e transição, que mostra respostas do sistema para os estímulos, é usado para expressar a especificação.
2. *Desenvolvimento incremental*. O software é particionado em incrementos desenvolvidos e validados separadamente, por meio do processo Cleanroom. Esses incrementos são especificados, com informações do cliente, em um estágio inicial do processo.
3. *Programação estruturada*. Somente um número limitado de construções abstratas de controle e de dados são usados. O processo de desenvolvimento de programa é um processo de refinamentos sucessivos da especificação. Um número limitado de construções é usado e o objetivo é transformar sistematicamente a especificação para criar o código de programa.
4. *Verificação estática*. O software desenvolvido é verificado estaticamente por meio de inspeções rigorosas de software. Não há processo de teste de módulo ou unitário para os componentes de código.
5. *Testes estatísticos de sistema*. O incremento de software integrado é testado estatisticamente, conforme explicado no Capítulo 24, para determinar sua confiabilidade. Esses testes estatísticos são baseados em um perfil operacional desenvolvido em paralelo com a especificação de sistema conforme mostrado na Figura 22.6.

Há três equipes envolvidas quando o processo Cleanroom é usado para desenvolvimento de sistema de grande porte:

1. *A equipe de especificação*. Esse grupo é responsável pelo desenvolvimento e pela manutenção da especificação de sistema. A equipe produz especificações orientadas ao cliente (a definição de requisitos de usuário) e especificações matemáticas para a verificação. Em alguns casos, quando a especificação está completa, a equipe de especificação é também responsável pelo desenvolvimento.
2. *A equipe de desenvolvimento*. Essa equipe tem a responsabilidade pelo desenvolvimento e pela verificação do software. O software não é executado durante o processo de desenvolvimento. Uma abordagem estruturada e formal é usada para verificação baseada em inspeção de código, complementada com argumentos de correção.
3. *A equipe de certificação*. Essa equipe é responsável pelo desenvolvimento de um conjunto de testes estatísticos para exercitar o software depois que ele foi desenvolvido. Os testes baseiam-se em especificação formal. O desenvolvimento de casos de teste é realizado em paralelo com o desenvolvimento de software. Os casos de teste são usados para certificar a confiabilidade de software. Os modelos de crescimento de confiabilidade (Capítulo 24) podem ser usados para decidir quando parar os testes.

Figura 22.6 Processo de desenvolvimento Cleanroom.



O uso da abordagem Cleanroom tem conduzido geralmente a um software com muito poucos erros. Cobb e Mills explicam vários projetos de desenvolvimento Cleanroom bem-sucedidos que tiveram uma baixa taxa de falhas uniforme em sistemas entregues (Cobb e Mills, 1990). Os custos desses projetos foram comparáveis com outros projetos que usaram técnicas de desenvolvimento convencionais.

A abordagem para desenvolvimento incremental no processo Cleanroom é entregar a funcionalidade crítica do cliente em incrementos mais cedo. As funções de sistema menos importantes são incluídas em incrementos mais tarde. O cliente, portanto, tem a oportunidade de tentar usar esses incrementos críticos antes que o sistema todo tenha sido entregue. Se problemas de requisitos forem descobertos, o cliente repassará essas informações para a equipe de desenvolvimento e solicitará um novo release do incremento.

Como na *extreme programming*, isso significa que as funções mais importantes do cliente recebem maior validação. Conforme novos incrementos são desenvolvidos, eles são combinados com os incrementos existentes e o sistema integrado é testado. Portanto, os incrementos existentes são novamente testados com novos casos de testes à medida que novos incrementos de sistema são adicionados.

A inspeção rigorosa de programa é uma parte fundamental do processo Cleanroom. Um modelo de estado do sistema é produzido como especificação de sistema. Esse modelo é refinado por meio de uma série de modelos de sistema mais detalhados para um programa executável. A abordagem usada para desenvolvimento baseia-se em transformações bem definidas que tentam preservar a correção, a cada transformação, para uma representação mais detalhada. A cada estágio, a nova representação é inspecionada e argumentos matematicamente rigorosos são desenvolvidos demonstrando que a saída da transformação é consistente com sua entrada.

Os argumentos matemáticos usados no processo Cleanroom não são, entretanto, provas formais de correção. Provas matemáticas formais de que um programa está correto em relação à sua especificação são muito dispendiosas para se desenvolver. Elas dependem do uso de conhecimento de semânticas formais da linguagem de programação para construir teorias que relacionem o programa e sua especificação formal. Essas teorias devem, portanto, ser provadas matematicamente, muitas vezes com a assistência de programas provadores de teorema, grandes e complexos. Devido a seu alto custo e às habilidades especializadas necessárias, as provas são geralmente desenvolvidas somente para aplicações mais críticas de segurança ou de proteção.

A inspeção e a análise formal foram consideradas muito eficientes no processo Cleanroom. A vasta maioria dos defeitos é descoberta antes da execução e não é introduzida no desenvolvimento de software. Linger (Linger, 1994) relata que, em média, somente 2,3 defeitos por mil linhas de código-fonte foram descobertos durante o teste em projetos Cleanroom. Os custos globais de desenvolvimento não aumentam, pois é necessário menos esforço para testar e reparar o software desenvolvido.

Selby et al., (Selby, et al., 1987), usando estudantes como desenvolvedores, conduziram um experimento que comparava desenvolvimento Cleanroom com técnicas convencionais. Eles constataram que a maioria das equipes poderia usar com sucesso o método Cleanroom. Os programas produzidos eram de qualidade maior do que os desenvolvidos por meio de técnicas tradicionais — o código-fonte tinha mais comentários e uma estrutura mais simples. A maioria das equipes Cleanroom cumpriu o cronograma de desenvolvimento.

O desenvolvimento Cleanroom funciona quando é praticado por engenheiros habilidosos e comprometidos. Relatos de sucesso da abordagem Cleanroom no setor vieram, em sua maioria, embora não exclusivamente, de pessoas já comprometidas com ela. Não sabemos se esse processo pode ser transferido eficientemente para outros tipos de organizações de desenvolvimento de software. Essas organizações podem ter engenheiros menos comprometidos e menos habilidosos. A transferência da abordagem Cleanroom ou, de fato, de qualquer outra abordagem na qual métodos formais são usados para organizações tecnicamente menos avançadas ainda representa um desafio.

PONTOS-CHAVE

- A verificação e validação não são a mesma coisa. Verificação se destina a mostrar que um programa atende à sua especificação. A validação se destina a mostrar que o programa realiza o que o usuário necessita.
- Os planos de teste devem incluir uma descrição de itens a serem testados: o cronograma de teste, os procedimentos para gerenciamento do processo de teste, o hardware e os requisitos de software, além de quaisquer outros problemas de teste que possam surgir.
- As técnicas de verificação estática envolvem o exame e a análise de código-fonte de programa para detecção de erros. Elas devem ser usadas com o teste de programa como parte do processo de V & V.
- As inspeções de programa são eficientes para encontrar erros de programa. O objetivo de uma inspeção é localizar defeitos. Um checklist de defeitos deve dirigir o processo de inspeção.

- Em um programa de inspeção, uma equipe pequena verifica sistematicamente o código. Membros de equipe incluem um líder da equipe ou moderador, o autor do código, um leitor que apresenta o código durante a inspeção e um testador que considera o código sob a perspectiva de teste.
- Os analisadores estáticos são ferramentas de software que processam um código-fonte de programa e chamam atenção para anomalias, como seções de código não usadas e variáveis não iniciadas. Essas anomalias podem ser resultantes de defeitos no código.
- O desenvolvimento de software Cleanroom conta com técnicas estáticas para verificação de programa e testes estatísticos para certificação de confiabilidade de sistema. Tem sido bem-sucedido na produção de sistemas com um alto nível de confiabilidade.

LEITURAS SUGERIDAS

Software quality assurance: from theory to implementation. Este livro oferece uma boa leitura de formação geral em verificação e validação, com um capítulo particularmente bom sobre revisões e inspeções. (D. Galin. 2004, Addison-Wesley.)

"Software inspection". Edição especial de um jornal com uma série de artigos sobre inspeção de programa, entre eles uma explicação sobre o uso dessa técnica em desenvolvimento orientado a objetos. (*IEEE software*, 20(4), julho/agosto de 2003.)

"Software debugging, testing and verification". Este é um artigo geral sobre verificação e validação e um dos poucos artigos que considera tanto o teste como as técnicas de verificação estática. (B. Hailpern e P. Santhanam, *IBM Systems Journal*, 41(1), janeiro de 2002.)

Cleanroom software engineering: technology and process. Um bom livro sobre a abordagem Cleanroom com seções sobre as bases da técnica, o processo e um estudo de caso prático. (S. J. Powell, et al., 1999, Addison-Wesley.)

EXERCÍCIOS

22.1 Explique as diferenças entre verificação e validação e explique por que a validação é um processo particularmente difícil.

22.2 Explique por que não é necessário que um programa seja completamente livre de defeitos antes de ser entregue a seus clientes. Até que ponto o teste pode ser usado para validar que o programa está adequado a seu propósito?

22.3 O plano de teste no Quadro 22.1 foi projetado para sistemas sob encomenda que têm um documento de requisitos separado. Sugira como a estrutura de plano de teste poderia ser modificada para testar produtos de software fechados.

22.4 Explique por que as inspeções de programa são uma técnica eficiente para descobrir erros em um programa. Que tipos de erro são improváveis de serem descobertos por meio de inspeções?

22.5 Sugira por que uma organização com uma cultura competitiva e elitista encontraria provavelmente dificuldade em introduzir inspeções de programa como uma técnica de V & V.

22.6 Usando seu conhecimento de Java, C++, C ou alguma outra linguagem de programação, desenvolva um checklist de erros comuns (não erros de sintaxe) que não poderiam ser detectados por um compilador, mas que poderiam ser detectados em uma inspeção de programa.

22.7 Faça uma lista de condições que poderiam ser detectadas por um analisador estático para Java, C++ ou outra linguagem de programação que você usa. Comente essa lista comparando-a com a lista fornecida na Tabela 22.2.

22.8 Explique por que pode ser de custo adequado usar métodos formais no desenvolvimento de sistemas de software críticos de segurança. Por que você acha que alguns desenvolvedores desse tipo de sistema são contra o uso de métodos formais?

22.9 Um gerente decide usar os relatórios de inspeções de programa como entrada para o processo de avaliação de pessoal. Esses relatórios mostram quem fez e quem descobriu os erros de programa. Esse é um comportamento gerencial ético? Seria ético se o pessoal fosse informado previamente de que isso ocorreria? Que diferença poderia fazer no processo de inspeção?

22.10 Uma abordagem comumente adotada para um teste de sistema é testar o sistema até terminar o orçamento destinado ao teste e depois entregar o sistema para os clientes. Explique a ética dessa abordagem.

23

Teste de software

Objetivos

O objetivo deste capítulo é descrever os processos de teste de software e apresentar algumas técnicas de teste. Após ler este capítulo, você compreenderá:

- as distinções entre teste de validação e teste de defeitos;
- os princípios de teste de sistemas e teste de componentes;
- três estratégias que podem ser usadas para gerar casos de teste de sistema;
- as características essenciais de ferramentas de software que apóiam a automação de testes.

Conteúdo

23.1 Teste de sistemas
23.2 Teste de componentes
23.3 Projeto de casos de teste
23.4 Automação de testes

No Capítulo 4 expliquei um processo geral de testes que começou com testes individuais de unidades de programa, como funções ou objetos. Estes eram integrados a subsistemas e sistemas, e as interações entre essas unidades eram testadas. Finalmente, depois da entrega do sistema, o cliente pode realizar uma série de testes de aceitação para verificar se o sistema funciona conforme especificado.

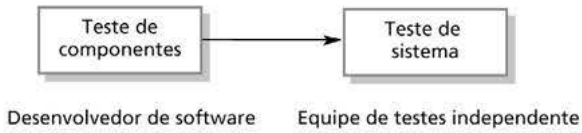
Esse modelo de processo de testes é apropriado para o desenvolvimento de sistemas de grande porte — porém para sistemas menores ou para sistemas desenvolvidos por meio de scripting ou reuso, existem alguns estágios distintos no processo. Uma visão mais abstrata de teste de software é mostrada na Figura 23.1. As duas atividades fundamentais são testes de componentes (teste de partes do sistema) e teste de sistema (teste do sistema como um todo).

O objetivo do estágio de testes de componente é descobrir defeitos por meio de testes de componentes individuais do programa. Esses componentes podem ser funções, objetos ou componentes reusáveis conforme descritos no Capítulo 19. Durante o teste de sistema, esses componentes são integrados para formar subsistemas ou o sistema completo. Nesse estágio, o teste de sistema deve enfocar a verificação de se o sistema atende aos requisitos funcionais e não funcionais e se não se comporta de maneira inesperada. Inevitavelmente, defeitos em componentes ignorados em testes anteriores serão descobertos durante o teste do sistema.

Conforme expliquei no Capítulo 22, o processo de teste de software tem duas metas distintas:

1. *Demonstrar ao desenvolvedor e ao cliente que o software atende aos requisitos.* Para software sob encomenda, isso significa que deve haver pelo menos um teste para cada requisito

Figura 23.1
Fases de teste.



contido nos documentos de usuário e de sistema. Para produtos de software genéricos, isso significa que deve haver testes para todas as características de sistema que serão incorporadas ao release do produto. Conforme explicado no Capítulo 4, alguns sistemas podem ter uma fase explícita de testes de aceitação, na qual o cliente verifica formalmente que o sistema entregue está em conformidade com sua especificação.

2. *Descobrir falhas ou defeitos no software que apresenta comportamento incorreto, não desejável ou em não conformidade com sua especificação.* O teste de defeitos está relacionado à remoção de todos os tipos de comportamentos indesejáveis de sistema, como travamentos, interações indesejáveis com outros sistemas, cálculos incorretos e corrompimento de dados.

A primeira meta conduz ao teste de validação, no qual você espera que o sistema seja executado corretamente em um dado conjunto de casos de teste que refletem o uso esperado do sistema. A segunda meta conduz ao teste de defeitos, no qual são projetados casos de teste para expor defeitos. Os casos de teste podem ser propositalmente obscuros e não precisam refletir como o sistema é usado normalmente. Para o teste de validação, um teste bem-sucedido é aquele em que o sistema funciona corretamente. Para o teste de defeitos, um teste bem-sucedido é o que expõe um defeito que causa o funcionamento incorreto do sistema.

Os testes não podem demonstrar que um software é livre de defeitos ou que ele se comportará conforme especificado em todas as circunstâncias. É sempre possível que um teste ignorado possa descobrir mais problemas no sistema. Como declarou eloqüentemente Edsger Dijkstra (Dijkstra, et al., 1972), uma personalidade notável no início do desenvolvimento de engenharia de software, "Os testes podem somente mostrar a presença de erros, não sua ausência".

Portanto, a meta do teste de software é convencer os desenvolvedores e clientes do sistema de que o software é bom o suficiente para o uso operacional. O teste é um processo voltado a atingir a confiabilidade do software.

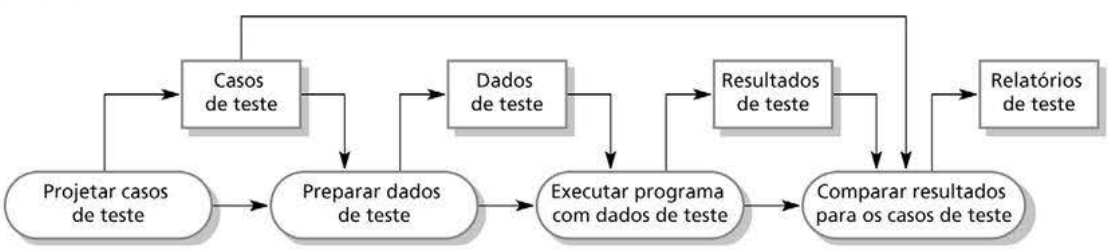
Um modelo geral do processo de teste é mostrado na Figura 23.2. Casos de teste são especificações de entradas para o teste e as saídas esperadas do sistema mais a declaração do que está sendo testado. Dados de teste são entradas preparadas para o teste de sistema. Dados de teste podem, muitas vezes, ser gerados automaticamente. A geração automática de casos de teste é impossível. A saída dos testes pode somente ser prevista por pessoas que compreendem o que o sistema deve fazer.

Testes exaustivos, nos quais cada seqüência possível de execução do programa é testada, são impossíveis. Os testes têm, portanto, de ser baseados em um subconjunto de casos possíveis de teste. De preferência, as empresas de software devem ter políticas para a escolha desse subconjunto em vez de deixar isso para a equipe de desenvolvimento. Essas políticas poderiam ser baseadas em políticas gerais de teste, como uma política na qual todas as declarações devem ser executadas pelo menos uma vez. De modo alternativo, as políticas de teste podem ser baseadas na experiência de uso do sistema e podem enfocar o teste das características do sistema que esteja operacional. Por exemplo:

1. Todas as funções do sistema acessadas por meio de menus devem ser testadas.
2. Combinações de funções (por exemplo, formatação de texto) acessadas por meio dos mesmos menus devem ser testadas.
3. Todas as funções devem ser testadas com entradas corretas e incorretas no local de entrada do usuário.

A experiência com os principais produtos de software, como processadores de texto e planilhas, deixa claro que diretrizes semelhantes têm sido usadas durante o teste de produtos. Quando características de software são usadas isoladamente, eles operam normalmente. Os problemas ocorrem, como explica Whittaker (Whittaker, 2002), quando combinações de caracte-

Figura 23.2 Modelo de processo de testes de software.



rísticas não são testadas conjuntamente. Ele apresenta o exemplo de que isso ocorre em um processador de texto que usa rodapés com layout de várias colunas, o que apresenta layout incorreto do texto.

Como parte do processo de planejamento de V & V, os gerentes devem tomar decisões sobre quem deve ser o responsável pelos diferentes estágios de teste. Para a maioria dos sistemas, os programadores têm a responsabilidade de testar os componentes que desenvolveram. Uma vez que isso é concluído, a tarefa é controlar uma equipe para integrar os módulos de diferentes desenvolvedores, construir o software e testar o sistema como um todo. Para sistemas críticos, um processo mais formal pode ser usado, no qual testadores independentes são responsáveis por todos os estágios do processo de teste. No teste de sistemas críticos, os testes são desenvolvidos separadamente e mantidos registros detalhados dos resultados do teste.

Testes de componente realizados por desenvolvedores geralmente baseiam-se no entendimento intuitivo de como os componentes devem operar. Testes de sistema, contudo, devem basear-se em uma especificação escrita do sistema. Ela pode ser uma especificação detalhada de requisitos de sistema, conforme explicado no Capítulo 6, ou pode ser uma especificação de características de alto nível orientada a usuários a serem implementadas no sistema. Uma outra equipe normalmente é responsável pelo teste de sistema. Conforme explicado no Capítulo 4, a equipe de testes de sistema trabalha com base em requisitos de usuário e de sistema para desenvolver os planos de testes de sistema (veja a Figura 4.10).

A maioria das discussões sobre testes começa com testes de componentes que passam a testes de sistema. Propositalmente, inverti a ordem da explicação neste capítulo, pois cada vez mais o desenvolvimento de software envolve a integração de componentes reusáveis, configuração e adaptação de software existente para atender aos requisitos especificados. Todos os testes, nesses casos, são testes de sistema, e não existe um processo separado de testes de componentes.

23.1 Teste de sistemas

O teste de sistema envolve a integração de dois ou mais componentes que implementam funções ou características do sistema e depois o teste desse sistema integrado. Em um processo de desenvolvimento iterativo, o teste de sistema concentra-se no teste de um incremento que será entregue ao cliente; em um processo em cascata, o teste de sistema concentra-se no teste de todo o sistema.

Para a maioria dos sistemas complexos, existem duas fases distintas de teste do sistema:

1. *Testes de integração,* nos quais a equipe de testes deve acessar o código-fonte do sistema. Quando um problema é descoberto, a equipe de integração tenta encontrar a origem do problema e identificar os componentes que devem ser depurados. Os testes de integração geralmente estão relacionados à descoberta de defeitos no sistema.
2. *Testes de releases,* nos quais uma versão do sistema, que poderia ser liberada aos usuários, é testada. Aqui, a equipe de testes concentra-se em validar se o sistema atende aos requisitos e em assegurar que o sistema é confiável. O teste de releases é normalmente um teste 'caixa-preta' no qual a equipe de testes concentra-se em demonstrar se o sistema funciona adequadamente ou não. Os problemas são reportados à equipe de desenvolvimento, cujo trabalho é depurar o programa. Os clientes são envolvidos no teste de releases que, algumas vezes, é chamado de *teste de aceitação.* Se o release for bom o suficiente, o cliente poderá aceitá-lo para seu uso.

Fundamentalmente, você pode pensar no teste de integração como o teste de sistemas incompletos compostos de clusters e agrupamentos de componentes de sistema. O teste de releases concentra-se em testar o release do sistema destinado à entrega aos clientes. Naturalmente, eles se sobrepõem, especialmente quando é usado o desenvolvimento incremental e o sistema a ser liberado está incompleto. Geralmente, a prioridade no teste de integração é descobrir defeitos no sistema e a prioridade no teste de sistemas é validar se o sistema atende aos requisitos. Contudo, na prática, ocorre algum teste de validação e algum teste de defeitos durante a execução desses processos.

23.1.1 Teste de integração

O processo de integração do sistema envolve a construção de um sistema com base em seus componentes (veja o Capítulo 29) e o teste do sistema resultante dos problemas ocorridos nas interações entre componentes. Os componentes integrados podem ser comerciais, reusáveis adaptados a determinado sistema ou componentes novos desenvolvidos. Para muitos sistemas de grande porte, provavelmente são usados os três tipos de componentes. O teste de integração verifica se esses componentes funcionam realmente em conjunto, se são chamados corretamente e se transferem dados corretos no tempo correto por meio de suas interfaces.

A integração do sistema envolve a identificação de clusters de componentes que realizam alguma funcionalidade do sistema, e a integração desses componentes pela adição de códigos que fazem com que eles trabalhem em conjunto. Algumas vezes, um esqueleto geral do sistema é desenvolvido primeiro, e os componentes são adicionados a ele. Essa estratégia é

chamada de *integração top-down*. Como alternativa, você pode integrar componentes de infra-estrutura que fornecem serviços comuns, como acesso à rede e ao banco de dados, e, em seguida, adicionar os componentes funcionais. Essa estratégia é chamada de *integração bottom-up*. Na prática, em muitos sistemas, a estratégia de integração é uma combinação dessas estratégias, com componentes de infra-estrutura e componentes funcionais adicionados em incrementos. Tanto na integração top-down quanto na bottom-up, você geralmente deve desenvolver códigos adicionais para simular outros componentes e permitir que o sistema funcione.

O problema principal durante o teste de integração é a localização de erros. Existem interações complexas entre os componentes do sistema e, quando uma saída anômala é descoberta, você pode ter dificuldades em identificar onde ocorreu o erro. Para tornar a localização de erros mais fácil, use uma abordagem incremental para integração e teste do sistema. Inicialmente, deve-se integrar uma configuração mínima do sistema e testá-lo. Em seguida, são adicionados componentes a essa configuração mínima e novos testes são executados.

No exemplo mostrado na Figura 23.3, A, B, C e D são componentes e T1 a T5 referem-se aos testes de características incorporadas ao sistema. T1, T2 e T3 são executados inicialmente no sistema composto dos componentes A e B (sistema mínimo). Se esses testes revelarem defeitos, estes serão corrigidos. O componente C é integrado e T1, T2 e T3 são repetidos para assegurar que não há interações inesperadas entre A e B. Se surgirem problemas nesses testes, isso provavelmente significa que eles são decorrentes da interação com o novo componente. A origem do problema é localizada, simplificando a identificação e correção do defeito. O conjunto de testes T4 é também executado no sistema. Finalmente, o componente D é integrado e testado por meio dos testes existentes e dos novos testes (T5).

Ao planejar a integração, você deve decidir a ordem de integração dos componentes. Em um processo como a *extreme programming*, o cliente é envolvido no processo de desenvolvimento e decide qual funcionalidade deve ser incluída em cada incremento do sistema. Assim, a integração do sistema é voltada às prioridades dos clientes. Em outras abordagens de desenvolvimento, quando componentes comerciais e os especialmente desenvolvidos são integrados, o cliente pode não estar envolvido e a equipe de integração decide as prioridades da integração.

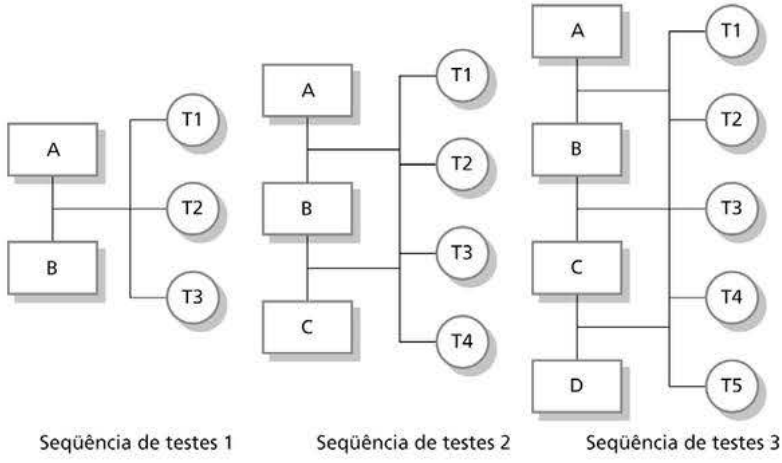
Nesses casos, uma boa regra prática é integrar os componentes que implementam a funcionalidade mais freqüentemente usada logo no início. Isso significa que os componentes mais usados devem passar pela maioria dos testes. Por exemplo, no sistema de biblioteca, LIBSYS, você deve começar pela integração do recurso de busca, de modo que, em um sistema mínimo, os usuários possam buscar documentos de que necessitam. Você deve, depois, adicionar a funcionalidade para permitir que os usuários possam baixar um documento e, progressivamente, adicionar os componentes que implementam outras características do sistema.

Naturalmente, a realidade não é tão simples quanto o modelo sugere. A implementação de características de sistema pode estar espalhada em uma série de componentes. Para testar uma nova característica, pode ser necessário integrar vários componentes diferentes. Os testes podem revelar erros na interação entre esses componentes individuais e em outras partes do sistema. A correção de erros pode ser difícil, pois um grupo de componentes que implementa a característica do sistema pode ser alterado. Além disso, a integração e o teste de um novo componente podem mudar o padrão das interações de componentes já testados. Podem ser revelados erros não expostos nos testes de configuração mais simples.

Esses problemas indicam que, quando um novo incremento é integrado, é importante executar os testes de incrementos anteriores novamente, bem como novos testes são necessários para verificar a nova funcionalidade do sistema. Executar um conjunto de testes existentes novamente é chamado de *teste de regressão*. Se o teste de regressão revelar problemas, você deve verificar se esses são problemas no incremento anterior que o novo incremento revelou ou se os problemas devem-se ao incremento de funcionalidade adicionado.

Figura 23.3

Testes de integração incremental.



O teste de regressão é claramente um processo dispendioso e é impraticável sem algum apoio automatizado. Na *extreme programming*, conforme explicado no Capítulo 17, todos os testes são escritos como códigos executáveis em que as entradas de teste e as saídas esperadas são especificadas e automaticamente verificadas. Quando é usado um framework de teste automatizado, como o JUnit (Massol e Husted, 2003), a nova execução dos testes pode ser automática. É um princípio básico da *extreme programming* que o conjunto completo de testes seja executado sempre que um novo código for integrado e que esse novo código não seja aceito até que todos os testes sejam executados com sucesso.

23.1.2 Teste de releases

O teste de releases é o processo de teste do release do sistema que será distribuído aos clientes. A meta principal desse processo é aumentar a confiança do fornecedor de que o sistema atende aos requisitos. Se sim, ele poderá ser liberado como um produto ou ser entregue ao cliente. Para demonstrar que o sistema atende aos requisitos, você deve mostrar que o sistema fornece a funcionalidade, o desempenho e a confiabilidade especificados, e que não apresenta falhas durante o uso normal.

O teste de releases geralmente é um processo de teste caixa-preta no qual os testes são derivados da especificação do sistema. O sistema é tratado como uma caixa-preta, cujo comportamento pode ser somente determinado por meio do estudo de suas entradas e as saídas relacionadas. Outro nome para isso é *teste funcional*, pois o testador concentra-se na funcionalidade, e não na implementação do software.

A Figura 23.4 ilustra o modelo de um sistema em teste de caixa-preta. O testador fornece as entradas para o componente ou o sistema e examina as saídas correspondentes. Se as saídas não forem as previstas (por exemplo, se as saídas estiverem no conjunto O_e), o teste detectou um problema com o software.

Quando são testados os releases do sistema, você deve tentar 'quebrar' o software escolhendo os casos de testes no conjunto I_e da Figura 23.4. Isto é, seu desafio deve ser selecionar entradas com alta probabilidade de gerar falhas de sistema (saídas do conjunto O_e). A experiência anterior a respeito de quais provavelmente serão os testes de defeitos bem-sucedidos e as diretrizes de teste ajudarão nessa escolha.

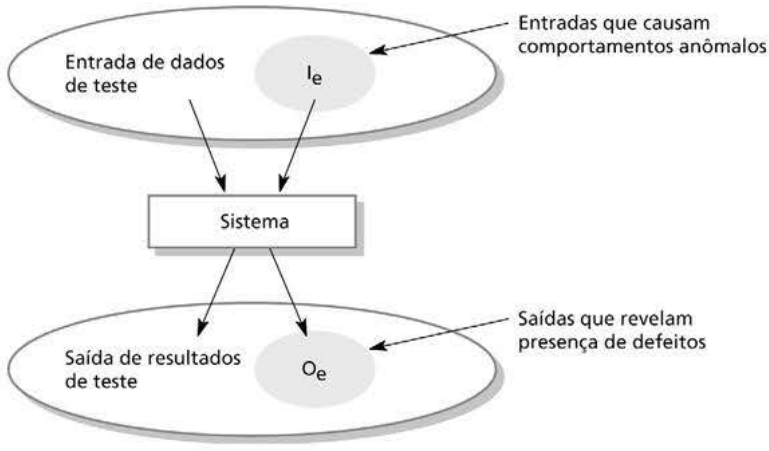
Autores como Whittaker (Whittaker, 2002) englobaram suas experiências com testes em um conjunto de diretrizes que aumentam a probabilidade de que os testes de defeitos sejam bem-sucedidos. Alguns exemplos dessas diretrizes são:

1. Escolher entradas que forcem o sistema a gerar todas as mensagens de erros.
2. Projetar entradas que causem overflow dos buffers.
3. Repetir a mesma entrada ou série de entradas várias vezes.
4. Forçar a geração de saídas inválidas.
5. Forçar os resultados de cálculos a serem muito grandes ou muito pequenos.

Para validar que o sistema atende a seus requisitos, a melhor abordagem a ser usada é o teste baseado em cenários, no qual é criado um número de cenários, e casos de teste são criados com base nesses cenários. Por exemplo, o seguinte cenário pode descrever como o sistema de biblioteca LIBSYS, explicado nos capítulos anteriores, pode ser usado:

Uma estudante na Escócia, que estuda História americana, recebeu a tarefa de escrever um artigo sobre 'A mentalidade provinciana no Oeste norte-americano de 1840 a 1880'. Para fazer isso, ela necessita encontrar fontes de várias

Figura 23.4
Teste caixa-preta.



bibliotecas. Ela entra no sistema LIBSYS e usa o recurso de busca para descobrir se pode acessar os documentos originais da época. Descobre fontes em várias bibliotecas de universidades dos EUA e baixa cópias desses documentos. Contudo, para determinado documento, ela precisa da confirmação de sua universidade de que é uma aluna legítima e que o uso do documento destina-se a fins não comerciais. A estudante, então, usa o recurso do LIBSYS para solicitar tal confirmação e registra sua solicitação. Caso a confirmação seja concedida, o documento será baixado para o servidor, registrado na biblioteca e impresso. Ela receberá uma mensagem do LIBSYS informando sobre o recebimento de uma mensagem de e-mail quando o documento impresso estiver disponível.

Com base nesse cenário, é possível criar uma série de testes que podem ser aplicados ao sistema LIBSYS proposto:

1. Testar o mecanismo de login usando informações corretas e incorretas para verificar se os usuários válidos são aceitos e os usuários inválidos, rejeitados.
2. Testar o recurso de busca usando consultas de fontes conhecidas para verificar se o mecanismo realmente localiza os documentos.
3. Testar o recurso de apresentação de sistema para verificar se as informações sobre documentos são apresentadas apropriadamente.
4. Testar o mecanismo de solicitação de permissão para baixar documentos.
5. Testar a resposta de e-mail que indica que o documento baixado está disponível.

Para cada um desses testes, você deve projetar um conjunto de testes que inclua entradas válidas e inválidas e que gere saídas válidas e inválidas. Você deve também organizar testes baseados em cenários de modo que os cenários mais prováveis sejam testados primeiro e os cenários incomuns ou excepcionais sejam considerados mais tarde, assim dedicando seus esforços às partes mais usadas do sistema.

Se você usou casos de uso para descrever os requisitos do sistema, esses casos de uso e os diagramas de seqüência associados podem ser a base dos testes do sistema. Os casos de uso e os diagramas de seqüência podem ser usados durante os testes de integração e de releases. Como ilustração, uso um exemplo do sistema de estação meteorológica descrito no Capítulo 14.

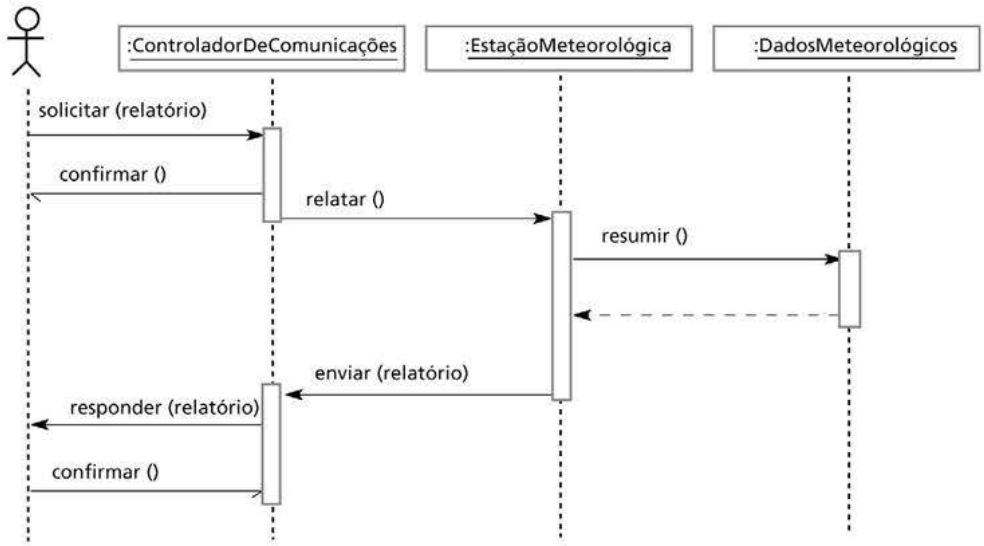
A Figura 23.5 mostra a seqüência de operações na estação meteorológica quando ela responde a uma solicitação de coleta de dados para o sistema de mapeamento. Você pode usar esse diagrama para identificar operações que serão testadas e para ajudar no projeto de casos de teste. Portanto, a emissão de uma requisição de um relatório resultará na execução do seguinte thread de métodos:

CommsController:solicitar → WeatherStation:relatar → WeatherData:resumir

O diagrama de seqüência pode também ser usado para identificar entradas e saídas que devem ser criadas para o teste:

1. A entrada de solicitação de um relátorio deve ter um acknowledge associado e um relatório deve ser retornado no final. Durante o teste, devem ser criados dados resumidos que possam ser usados para verificar se o relatório está corretamente organizado.

Figura 23.5 Diagrama de seqüência de coleta de dados meteorológicos.



2. Uma solicitação de entrada de um relatório para **WeatherStation** resulta na geração de um relatório resumido. Você pode testar essa situação isoladamente por meio da criação de dados brutos correspondentes ao resumo preparado para o teste de **CommsController** e da verificação de se o objeto **WeatherStation** produz corretamente esse resumo.
3. Esses dados brutos são também usados no teste do objeto **WeatherData**.

Naturalmente, simplifiquei o diagrama de seqüência na Figura 23.5, de modo que ele não mostra exceções. Um teste de cenário completo deve levar em conta essas exceções e assegurar que os objetos tratam as exceções corretamente.

23.1.3 Teste de desempenho

Após o sistema ter sido completamente integrado, é possível testá-lo em relação às propriedades emergentes (veja o Capítulo 2), como desempenho e confiabilidade. Os testes de desempenho devem ser projetados para assegurar que o sistema pode operar na carga necessária. Isso envolve, geralmente, o planejamento de uma série de testes em que a carga é constantemente aumentada até que o desempenho se torne inaceitável.

Como em outros tipos de teste, o teste de desempenho concentra-se tanto em demonstrar que o sistema atende aos requisitos como também em descobrir problemas e defeitos no sistema. Para testar se os requisitos de desempenho estão sendo atingidos, você pode precisar construir um perfil operacional. Um perfil operacional é um conjunto de testes que reflete uma combinação real de trabalho a que o sistema será submetido. Portanto, se 90% das transações de um sistema forem do tipo A, 5% forem do tipo B e o restante dos tipos C, D e E, você deverá projetar o perfil operacional de modo que a maioria dos testes seja do tipo A. Por outro lado, você não obterá um teste preciso de desempenho operacional do sistema. Explico os perfis operacionais e seu uso em testes de confiabilidade no Capítulo 24.

É claro que essa abordagem não é necessariamente a melhor para teste de defeitos. Conforme explicado mais adiante, experiências mostraram que uma maneira eficiente de descobrir defeitos é projetar testes com base nos limites do sistema. No teste de desempenho, isso significa estressar o sistema (por isso o nome *teste de estresse*) por meio de demandas fora dos limites do projeto do software.

Por exemplo, um sistema de processamento de transações pode ser projetado para processar até 300 transações por segundo; um sistema operacional pode ser projetado para lidar com até 1.000 terminais distintos. O teste de estresse estende os testes para além da carga máxima projetada do sistema até que ele falhe. Esse tipo de teste tem duas funções:

1. Testar o comportamento de falha do sistema. Podem surgir circunstâncias por meio de uma combinação de eventos em que a carga imposta ao sistema exceda a carga máxima projetada. Nessas circunstâncias, é importante que a falha do sistema não cause o corrompimento de dados ou perda inesperada de serviços de usuário. O teste de estresse verifica se a sobrecarga causa uma 'falha leve' em vez de um colapso quando o sistema é submetido a essa carga.
2. Estressar o sistema e causar o surgimento de defeitos que não seriam normalmente descobertos. Embora se possa argumentar que não é provável que esses defeitos causem falhas durante o uso normal, pode haver combinações incomuns de circunstâncias normais a que o teste de estresse responda.

O teste de estresse é particularmente relevante para sistemas distribuídos baseados em uma rede de processadores. Esses sistemas freqüentemente exibem degradações graves ao serem submetidos a pesadas cargas de demanda. A rede é inundada com dados de coordenação que os diferentes processos devem trocar e, assim, os processos tornam-se cada vez mais lentos enquanto aguardam os dados provenientes de outros processos.

23.2 Teste de componentes

O teste de componentes (algumas vezes chamado de *teste de unidade*) é o processo de teste de componentes individuais do sistema. Este é um processo de teste de defeitos e, portanto, sua meta é expor defeitos nesses componentes. Conforme expliquei na introdução, na maioria dos sistemas, os desenvolvedores são os responsáveis pelo teste de componentes.

Existem diferentes tipos de componentes que podem ser testados nesse estágio:

1. Funções ou métodos individuais de um objeto.
2. Classes de objeto com vários atributos e métodos.
3. Componentes compostos que constituem diferentes objetos ou funções. Esses componentes compostos têm uma interface definida usada para acessar sua funcionalidade.

As funções ou métodos individuais são os tipos mais simples de componentes e seus testes são um conjunto de chamadas dessas rotinas com diferentes parâmetros de entrada. Você pode usar as abordagens para o projeto de casos de teste, explicadas na próxima seção, para projetar testes de funções ou métodos.

Ao testar classes de objeto, você deve projetar seus testes de modo que abranjam todas as características do objeto. Dessa maneira, os testes devem incluir:

1. Testes isolados de todas as operações associadas ao objeto.
2. Atribuir e interrogar todos os atributos associados ao objeto.
3. O exercício do objeto em todos os estados possíveis. Isso significa que todos os eventos que causam mudanças de estado no objeto devem ser simulados.

Considere, por exemplo, a estação meteorológica do Capítulo 14, cuja interface é mostrada na Figura 23.6. Ela tem somente um atributo, que é seu identificador. O identificador é uma constante definida quando a estação meteorológica é instalada. Você precisa, assim, somente de um teste para verificar se o identificador foi definido. É preciso definir os casos de teste para **relatarClima**, **calibrar**, **testar**, **iniciar** e **desativar**. De preferência, você deve testar esses métodos isoladamente, porém, em alguns casos, algumas seqüências de teste são necessárias. Por exemplo, para testar o método **desativar**, é necessário ter executado o método **iniciar**.

Para testar os estados da estação meteorológica, é usado o modelo de estados conforme mostrado na Figura 14.14. Usando esse modelo, você pode identificar as seqüências de transições de estados que devem ser testadas e definir seqüências de eventos que forçam essas transições. Em princípio, você deve testar todas as seqüências de transição de estado possíveis, embora, na prática, isso possa se tornar muito dispendioso. Exemplos de seqüências de estados a serem testadas na estação meteorológica incluem:

Desativado → Aguardando → Desativado
Aguardando → Calibrando → Testando → Transmitindo → Aguardando
Aguardando → Coletando → Aguardando → Resumindo → Transmitindo → Aguardando

Se você usa herança, torna-se mais difícil projetar testes de classes de objeto. Quando uma superclasse fornece operações herdadas por várias subclasses, todas essas subclasses devem ser testadas com todas as operações herdadas. O motivo disso é que a operação herdada pode assumir hipóteses sobre outras operações e atributos, os quais podem ter sido alterados quando herdados. Da mesma maneira, quando uma operação de superclasse for sobreposta, a operação sobrescrita deverá ser testada.

A noção de classes de equivalência, explicada na Seção 23.3.2, pode ser aplicada a classes de objeto. Testes na mesma classe de equivalência podem ser os que usam os mesmos atributos de objetos. Portanto, as classes de equivalência que iniciam, acessam e atualizam todos os atributos de classes de objeto devem ser identificadas.

23.2.1 Teste de interfaces

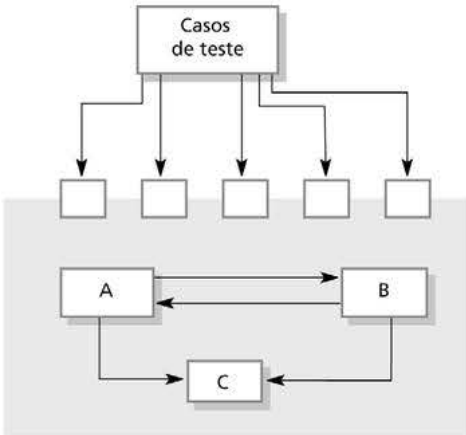
Muitos componentes em um sistema não são funções ou objetos simples, mas sim componentes compostos constituídos de vários objetos que interagem. Conforme explicado no Capítulo 19, que aborda a engenharia de software baseada em componentes, a funcionalidade desses componentes é acessada por meio de suas interfaces definidas. O teste dos componentes compostos concentra-se principalmente em se a interface de componente se comporta de acordo com sua especificação.

A Figura 23.7 mostra esse processo de teste de interfaces. Suponha que os componentes A, B e C foram integrados para formar um componente ou um subsistema maior. Os casos de teste não são aplicados nos componentes individuais, mas sim na interface do componente composto, formado pela combinação dos componentes.

O teste de interfaces é particularmente importante para o desenvolvimento orientado a objetos e baseado em componentes. Os objetos e os componentes são definidos por suas interfaces e podem ser reusados em combinação com outros componentes em diferentes sistemas. Erros de interface no componente composto não podem ser detectados por meio de testes de objetos ou componentes individuais. Os erros no componente composto podem surgir devido à interação entre suas partes.

Figura 23.6
Interface de objeto da estação meteorológica.

Figura 23.7
Teste de interface.



Existem diferentes tipos de interfaces entre os componentes de programa e, conseqüentemente, diferentes tipos de erros podem ocorrer:

1. *Interfaces de parâmetros.* Estas são interfaces em que dados ou, algumas vezes, referências a funções são passados de um componente para outro.
2. *Interfaces de memória compartilhada.* São interfaces em que um bloco de memória é compartilhado entre os componentes. Os dados são colocados na memória por um subsistema e recuperados dali por outros subsistemas.
3. *Interfaces de procedimentos.* São interfaces em que um componente engloba um conjunto de procedimentos que podem ser chamados por outros componentes. Os objetos e os componentes reusáveis têm essa forma de interface.
4. *Interfaces de passagem de mensagem.* São interfaces em que um componente solicita um serviço de um outro componente passando uma mensagem para ele. Uma mensagem de retorno inclui os resultados da execução do serviço. Alguns sistemas orientados a objetos têm essa forma de interface, como ocorre em sistemas cliente–servidor.

Erros de interface são uma das formas mais comuns de erros em sistemas complexos (Lutz, 1993). Esses erros são divididos em três categorias:

1. *Mau uso de interface.* Um componente chama algum outro componente e comete um erro no uso de sua interface. Esse tipo de erro é particularmente comum com interfaces de parâmetros nas quais os parâmetros podem ser do tipo incorreto, podem ser passados em ordem incorreta ou uma quantidade incorreta de parâmetros pode ser passada.
2. *Mau entendimento da interface.* Um componente chamador perde a especificação da inteface do componente chamado e faz suposições sobre seu comportamento. O componente chamado não se comporta como esperado e isso causa um comportamento inesperado no componente chamador. Por exemplo, uma rotina de busca binária pode ser chamada com um vetor não ordenado para ser pesquisado. A busca falhará.
3. *Erros de timing.* Esses erros ocorrem em sistemas de tempo real que usam memória compartilhada ou uma interface de passagem de mensagens. O produtor e o consumidor de dados podem funcionar em velocidades diferentes. A menos que cuidados especiais sejam tomados no projeto de interface, o consumidor pode acessar dados desatualizados porque o produtor das informações não atualizou as informações de interface compartilhada.

O teste de defeitos de interface é difícil porque alguns defeitos de interface podem se manifestar somente sob condições incomuns. Por exemplo, digamos que um objeto implementa uma fila com uma estrutura de dados de comprimento fixo. Um objeto chamador pode supor que a fila é implementada como uma estrutura infinita de dados e pode não verificar o overflow de fila quando um item for inserido. Essa condição pode ser detectada durante os testes somente por meio do projeto de casos de teste que forçam a fila ao overflow provocando o corrompimento do comportamento do objeto de maneira detectável.

Um problema adicional pode surgir em função de interações dos defeitos em diferentes módulos ou objetos. Os defeitos em um objeto podem ser detectados quando algum outro objeto se comporta de modo inesperado. Por exemplo, um objeto pode chamar algum outro objeto para receber algum serviço e supor que a resposta esteja correta. Se houver má interpretação sobre o valor calculado, o valor retornado poderá ser válido, porém incorreto. Isso se manifestará apenas quando algum cálculo posterior for incorreto.

Algumas diretrizes gerais para teste de interfaces são:

1. Examine o código a ser testado e liste explicitamente cada chamada a componentes externos. Projete um conjunto de testes em que os valores dos parâmetros dos componentes externos estejam nos limites extremos de suas faixas. Será mais provável que esses valores extremos revelem inconsistências de interfaces.
2. Teste sempre a interface com ponteiros nulos onde os ponteiros são passados por meio de uma interface.
3. Projete testes que causem a falha do componente onde este é chamado por meio de uma interface de procedimento. Diferentes suposições a respeito de falhas é um dos mais comuns casos de mau entendimento de especificação.
4. Use o teste de estresse, conforme explicado na seção anterior, em sistemas de passagem de mensagens. Projete testes que gerem muito mais mensagens do que podem ocorrer na prática. Problemas de timing podem ser revelados dessa maneira.
5. Quando vários componentes interagem por meio de memória compartilhada, projete testes que variem a ordem na qual esses componentes são ativados. Esses testes podem revelar suposições implícitas feitas pelo programador a respeito da ordem na qual os dados compartilhados são produzidos e consumidos.

As técnicas de validação estáticas apresentam muitas vezes custos mais adequados do que os testes para descobrir erros de interface. Uma linguagem fortemente orientada a tipos, como Java, possibilita que muitos erros de interface sejam detectados pelo compilador. Quando uma linguagem mais fraca em termos de tipo de dados é usada, como C, um analisador estático, como Lint (veja o Capítulo 22), pode detectar erros de interface. Inspeções de programa podem se concentrar em interfaces de componentes e, em questões sobre o comportamento assumido da interface, ser aplicadas durante o processo de inspeção.

23.3 Projeto de casos de teste

O projeto de casos de teste é parte do teste de sistemas e de componentes, no qual você projeta os casos (entradas e saídas esperadas) que testam o sistema. A meta do processo de projeto de casos de teste é criar um conjunto de casos de teste eficazes para descobrir defeitos do programa e demonstrar que o sistema atende aos requisitos.

Para projetar um caso de teste, selecione uma característica do sistema ou do componente que você vai testar. Depois, selecione um conjunto de entradas para executar aquela característica, documente as saídas esperadas ou faixas de saídas e, quando possível, projete uma verificação automática que teste se as saídas reais e a saídas esperadas coincidem.

Existem várias abordagens que você pode usar para projetar casos de teste:

1. *Teste baseado em requisitos* no qual os casos de teste são projetados para testar os requisitos de sistema. Essa abordagem é uma das mais usadas no estágio de teste de sistema quando os requisitos de sistema geralmente são implementados por vários componentes. Identifique os casos de teste que possam demonstrar que o sistema atende a cada requisito.
2. *Teste de partições* no qual são identificadas partições de entrada e de saída e projetados testes de modo que o sistema processe as entradas de todas as partições e gere as saídas em todas as partições. As partições são grupos de dados com características em comum, como números negativos, todos os nomes com menos de 30 caracteres, todos os eventos que surgem pela escolha de itens de um menu etc.
3. *Teste estrutural* no qual é usado o conhecimento da estrutura do programa para projetar testes que exercitem todas as partes do programa. Essencialmente, ao testar um programa, você deve tentar executar cada declaração pelo menos uma vez. O teste estrutural ajuda a identificar casos de teste que podem tornar isso possível.

Em geral, quando se projetam casos de testes, você deve começar com testes do mais alto nível com base em requisitos e então, progressivamente, adicionar testes mais detalhados usando o teste de partições e o teste estrutural.

23.3.1 Teste baseado em requisitos

Um princípio geral da engenharia de requisitos, explicado no Capítulo 6, é que os requisitos devem ser testáveis. Ou seja, o requisito deve ser escrito de maneira que um conjunto de testes possa ser projetado para que um observador consiga verificar se o requisito foi satisfeito. Testes baseados em requisitos são, portanto, uma abordagem sistemática para projetar casos de teste nos quais cada requisito é considerado e um conjunto de testes é derivado para ele. O teste baseado em requisitos é um teste de validação em vez de um teste de defeitos — você está tentando demonstrar que o sistema tem seus requisitos adequadamente implementados.

Como exemplo, considere os requisitos para o sistema LIBSYS apresentado no Capítulo 6.

1. O usuário será capaz de procurar todo o conjunto inicial do banco de dados ou selecionar um subconjunto dele.
2. O sistema fornecerá visualizadores apropriados para que o usuário possa ler documentos no repositório de documentos.
3. A cada solicitação será alocado um identificador único (ORDER_ID) com o qual o usuário será capaz de fazer cópias para a área de armazenamento permanente.

Possíveis testes para os primeiros requisitos, presumindo que uma função de busca foi testada, são:

- Iniciar as buscas de usuário pelos itens cuja presença ou ausência são conhecidas, quando o conjunto de bancos de dados incluir um banco de dados.
- Iniciar as buscas de usuário pelos itens cuja presença ou ausência são conhecidas, quando o conjunto de bancos de dados incluir dois bancos de dados.
- Iniciar as buscas de usuário pelos itens cuja presença ou ausência são conhecidas, quando o conjunto de bancos de dados incluir mais de dois banco de dados.
- Selecionar um banco de dados do conjunto de bancos de dados e iniciar as buscas de usuário pelos itens cuja presença ou ausência são conhecidas.
- Selecionar mais de um banco de dados de um conjunto de banco de dados e iniciar as buscas pelos itens cuja presença ou ausência são conhecidas.

Com base nisso, é possível perceber que testar um requisito não significa escrever somente um único teste. Você deve, normalmente, escrever vários testes para assegurar que o requisito foi coberto.

Os testes para os outros requisitos do sistema LIBSYS podem ser desenvolvidos da mesma maneira. Para o segundo requisito, você deve escrever testes de documentos liberados de todos os tipos que possam ser processados pelo sistema e verificar se esses documentos são apresentados apropriadamente na tela. O terceiro requisito é mais simples. Para testá-lo, simule o envio de várias solicitações e, em seguida, verifique se o identificador da solicitação está presente na confirmação do usuário e se é único para cada caso.

23.3.2 Teste de partições

Os dados de entrada e os resultados de saída de um programa geralmente caem em classes diferentes com características comuns, como números positivos, números negativos e seleções de menu. Os programas normalmente se comportam de maneira comparável a todos os membros de uma classe. Ou seja, se você testar um programa que realiza um cálculo e requer dois números positivos, você espera que o programa se comporte da mesma maneira em relação a todos os números positivos.

Devido a esse comportamento equivalente, essas classes são algumas vezes chamadas de *partições de equivalência* ou *domínios* (Bezier, 1990). Uma abordagem sistemática para projetar casos de teste baseia-se na identificação de todas as partições de um sistema ou componente. Os casos de teste são projetados de modo que as entradas e as saídas se acomodem nessas partições. O teste de partições pode ser usado para projetar casos de teste para sistemas e componentes.

Na Figura 23.8, cada partição de equivalência é mostrada como uma elipse. Partições de equivalência de entrada são conjuntos de dados em que todos os membros do conjunto devem ser processados de modo equivalente. Partições de equivalência de saída são saídas de programa com características em comum, de maneira que elas possam ser consideradas como uma classe distinta. Você pode também identificar partições em que as entradas estão fora de outras partições escolhidas. Essas partições testam se o programa trata as entradas inválidas de maneira correta. Entradas válidas e inválidas também formam partições equivalentes.

Após identificar um conjunto de partições, você pode escolher casos de teste de cada uma dessas partições. Uma boa regra prática para seleção de casos de teste é escolher os casos de teste no limite das partições mais os casos próximos ao ponto médio da partição. A razão para isso é que os projetistas e os programadores tendem a considerar valores típicos de entradas ao desenvolverem um sistema. Esses valores são testados pela escolha do ponto médio da partição. Valores-limite são freqüentemente atípicos (por exemplo, zero pode se comportar de maneira diferente de números não negativos) e, portanto, ignorados pelos desenvolvedores. Muitas vezes ocorrem falhas de programa quando esses valores atípicos são processados.

As partições são identificadas por meio de especificações de programa ou documentação de usuário e também pela experiência, que permite prever classes de valores de entrada com probabilidade de detectar erros. Por exemplo, digamos que uma especificação de programa estabeleça que o programa aceite de 4 a 8 entradas, sendo estas números inteiros de cinco dígitos maiores do que 10.000. A Figura 23.9 mostra as partições para essa situação e valores possíveis de entradas de teste.

Para ilustrar a derivação de casos de teste, uso a especificação de um componente de busca mostrada na Figura 23.10. Esse componente busca uma seqüência de elementos para determinado elemento (key). Ele retorna a posição desse ele-

Figura 23.8
Particionamento de equivalência.

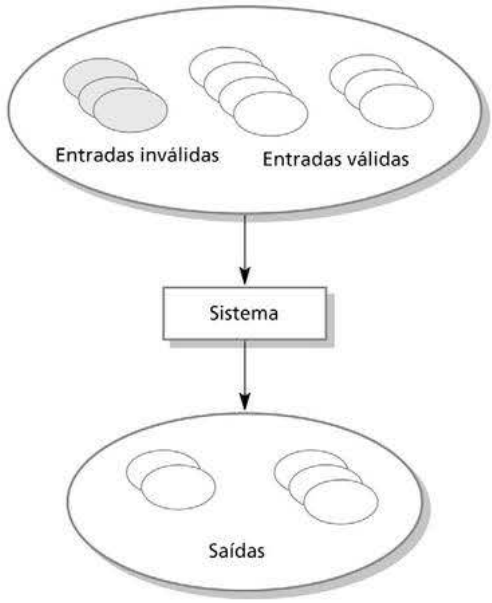


Figura 23.9
Partições de equivalência.

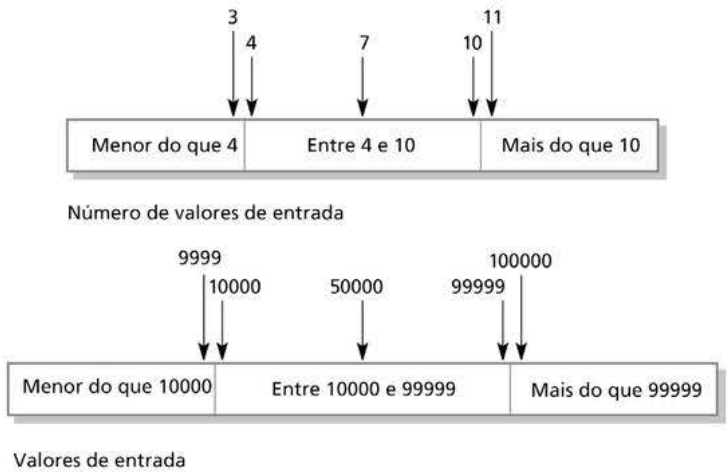


Figura 23.10
Especificação de uma rotina de busca.

```
procedimento Search (Key : ELEM ; T: SEQ of ELEM ;
Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
Precondição
    – a seqüência tem pelo menos um elemento
    T'FIRST <= T'LAST
Pós-condição
    – o elemento é encontrado e referenciado por L
    ( Found and T (L) = Key)
ou
    – o elemento não está na seqüência
    ( not Found and
    not (exists i, T'FIRST >= i <= T'LAST, T (i) = Key))
```

mento na seqüência. Especifiquei isso de maneira abstrata pela definição de precondições, que são verdadeiras antes que o componente seja chamado, e de pós-condições, que são verdadeiras após a execução. A precondição estabelece que a rotina de busca funcionará somente com seqüências que não estejam vazias. A pós-condição estabelece que a variável **Found** será definida se o elemento key estiver na seqüência. A posição do elemento key é o índice **L**. O valor de índice será indefinido se o elemento não estiver na seqüência.

Com base nessa especificação, você pode observar duas partições de equivalência:

1. Entradas, em que o elemento key é um membro da seqüência (**Found = true**).
2. Entradas, em que o elemento key não é um membro da seqüência (**Found = false**).

Quando você está testando programas com seqüências, vetores ou listas, existe uma série de diretrizes muitas vezes úteis para o projeto de casos de teste:

1. Testar software com seqüências que possuem um único valor. Os programadores, naturalmente, pensam em seqüências constituídas de vários valores e, algumas vezes, incluem essa suposição em seus programas. Conseqüentemente, o programa pode não funcionar apropriadamente quando submetido a uma seqüência de valor único.
2. Usar seqüências diferentes, de tamanhos diferentes, em testes diferentes. Isso diminui as chances de um programa com defeitos produzir acidentalmente uma saída correta devido a algumas características acidentais da entrada.
3. Derivar testes de modo que o primeiro, o médio e o último elementos da seqüência sejam acessados. Essa abordagem revela problemas nos limites das partições.

Baseadas nessas diretrizes, mais duas partições de equivalência podem ser identificadas:

1. A seqüência de entrada tem um único valor.
2. O número de elementos na seqüência de entrada é maior do que 1.

As partições adicionais, então, são identificadas pela combinação dessas partições — por exemplo, a partição em que o número de elementos na seqüência é maior do que 1 e o elemento não está na seqüência. A Tabela 23.1 mostra as partições que identifiquei para testar o componente de busca.

Um conjunto de possíveis casos de teste dessas partições também é mostrado na Tabela 23.1. Se o elemento key não está na seqüência, o valor de L será indefinido ('??'). A diretriz de seqüências diferentes de tamanhos diferentes deve ser usada nesses casos de teste.

O conjunto de valores de entradas usado para testar a rotina de busca não é exaustivo. A rotina poderá falhar se a seqüência de entrada incluir os elementos 1, 2, 3 e 4. Contudo, é razoável supor que se o teste falhar em detectar defeitos quando um membro de uma classe for processado, nenhum outro membro dessa classe identificará defeitos. Naturalmente, ainda podem existir defeitos. Algumas partições de equivalência podem não ter sido identificadas, podem ter sido cometidos erros na identificação da partição de equivalência ou os dados de teste podem ter sido preparados incorretamente.

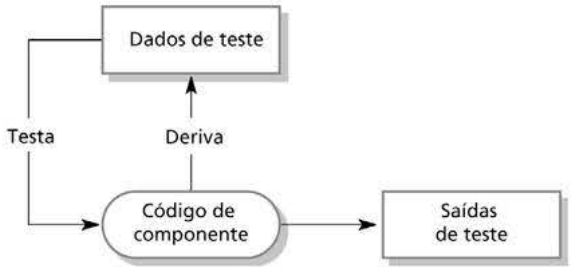
23.3.3 Teste estrutural

O teste estrutural (Figura 23.11) é uma abordagem para projetar casos de teste na qual os testes são derivados do conhecimento da estrutura e da implementação do software. Essa abordagem é, algumas vezes, chamada de teste 'caixa-branca', teste 'caixa-de-vidro' ou teste 'caixa-clara' para distingui-lo do teste caixa-preta.

Tabela 23.1 Partições equivalentes para rotina de busca

Seqüência	Elemento	
Valor único	Em seqüência	
Valor único	Fora da seqüência	
Mais de 1 valor	Primeiro elemento na seqüência	
Mais de 1 valor	Último elemento na seqüência	
Mais de 1 valor	Elemento médio na seqüência	
Mais de 1 valor	Fora da seqüência	
Seqüência de entradas (T)	**Chave (Key)**	**Saídas (Found, L)**
17	17	verdadeiro, 1
17	0	falso, ??
17, 29, 21, 23	17	verdadeiro, 1
41, 18, 9, 31, 30, 16, 45	45	verdadeiro, 7
17, 18, 21, 23, 29, 41, 38	23	verdadeiro, 4
21, 23, 29 33, 38	25	falso, ??

Figura 23.11
Teste estrutural.



O entendimento do algoritmo usado em um componente pode ajudar a identificar partições e casos de teste adicionais. Para ilustrar isso, implementei a especificação de rotina de busca (Figura 23.10) como uma rotina de busca binária (Figura 23.12). Naturalmente, a especificação tem precondições estritas. A seqüência é implementada como um vetor que deve ser ordenado e o valor do limite inferior do vetor deve ser menor do que o valor do limite superior.

Examinando o código da rotina de busca, você pode verificar que a busca binária envolve a divisão do espaço de busca em três partes. Cada uma dessas partes constitui uma partição de equivalência (Figura 23.13). Então, projetam-se os casos de teste onde o elemento key estiver nos limites de cada uma dessas partições.

Isso conduz a um conjunto de casos de testes atualizado para a rotina de busca, conforme mostrado na Tabela 23.2. Observe que modifiquei o vetor de entrada de modo que ele seja organizado em ordem crescente e adicionei testes onde o elemento key é adjacente ao ponto médio do vetor.

Figura 23.12
Implementação em Java de uma rotina de busca binária.

```
class BinSearch {

// Este é um encapsulamento de uma função de busca binária que
// considera um vetor (array) de objetos ordenados e uma chave (key)
// e retorna um objeto com 2 atributos denominados
// index – o valor do índice de vetor
// found – um valor booleano que indica se uma chave está ou não no vetor
// Um objeto é retornado, pois não é possível, em Java, passar tipos básicos
// por referência a uma função e, assim, retornar dois valores
// a chave (key) é -1, caso o elemento não seja encontrado

        public static void search ( int key, int [] elemArray, Result r )
        {
1.          int bottom = 0 ;
2.          int top = elemArray.length - 1 ;
            int mid ;
3.          r.found = false ;
4.          r.index = -1 ;
5.          while ( bottom <= top )
            {
6.              mid = (top + bottom) / 2 ;
7.              if (elemArray [mid] == key)
                {
8.                  r.index = mid ;
9.                  r.found = true ;
10.                   return ;
                } // if part
                else
                {
11.                 if (elemArray [mid] < key)
12.                     bottom = mid + 1 ;
                    else
13.                     top = mid - 1 ;
                }
            } //while loop
14.     } // search
} //BinSearch
```

Figura 23.13
Classes de equivalência de busca binária.

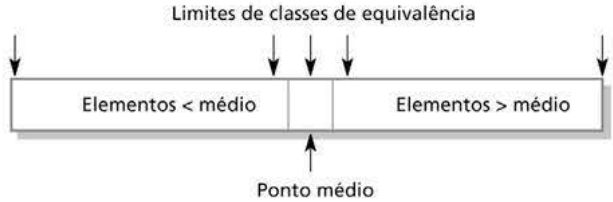


Tabela 23.2 Casos de teste para a rotina de busca

Vetor de entrada (T)	Elemento-chave (Key)	Saída (Found, L)
17	17	verdadeiro, 1
17	0	falso, ??
17, 21, 23, 29	17	verdadeiro, 1
9, 16, 18, 30, 31, 41, 45	45	verdadeiro, 7
17, 18, 21, 23, 29, 38, 41	23	verdadeiro, 4
17, 18, 21, 23, 29, 33, 38	21	verdadeiro, 3
12, 18, 21, 23, 32	23	verdadeiro, 4
21, 23, 29, 33, 38	25	falso, ??

23.3.4 Teste de caminho

O teste de caminho é uma estratégia de teste estrutural cujo objetivo é exercitar cada caminho independentemente da execução de um componente ou de um programa. Se cada caminho independente for executado, todas as declarações no componente terão sido executadas pelo menos uma vez. Além disso, todas as declarações condicionais são testadas para ambos os casos, verdadeira e falsa. Em um processo de desenvolvimento orientado a objetos, o teste de caminho pode ser usado no teste de métodos associados aos objetos.

O número de caminhos de um programa geralmente é proporcional a seu tamanho. Quando os módulos são integrados ao sistema, torna-se inviável usar técnicas de teste estrutural. As técnicas de teste de caminho são, portanto, muito mais usadas durante o teste de componentes.

O teste de caminho não testa todas as combinações possíveis de todos os caminhos do programa. Para quaisquer componentes, menos aqueles muito triviais sem loops, esse é um objetivo impossível. Existe um número infinito de combinações possíveis de caminhos em programas com loops. Mesmo quando todas as declarações são executadas pelo menos uma vez, podem ainda aparecer defeitos de programas quando determinados caminhos são combinados.

O ponto de partida para o teste de caminho é um fluxograma do programa. Este é um modelo esquemático de todos os caminhos do programa. Um fluxograma consiste de nós que representam decisões e linhas que mostram o fluxo de controle. O fluxograma é construído substituindo-se as declarações de controle de programa por diagramas equivalentes. Se não houver declarações 'goto' no programa, derivar seu fluxograma é um processo simples. Cada ramo em uma declaração condicional (*if-then-else* ou *case*) é mostrado como um caminho separado. Uma seta que retorna ao nó de condição denota um loop. Desenhei um fluxograma para o método de busca binária na Figura 23.14. Para tornar mais óbvia a correspondência entre o diagrama e o programa da Figura 23.12, mostrei cada declaração como um nó separado, sendo o número de nó correspondente a um número de linha no programa.

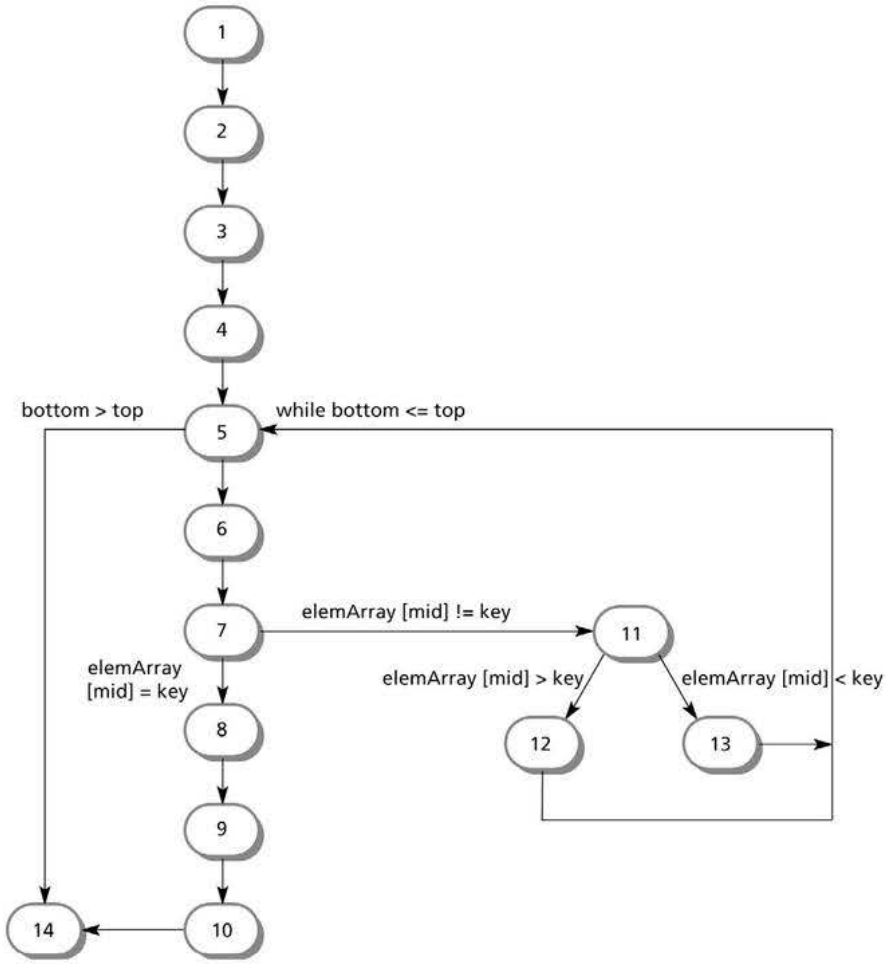
O objetivo do teste de caminho é assegurar que cada caminho independente do programa seja executado pelo menos uma vez. Um caminho independente do programa é aquele que atravessa pelo menos uma nova linha no fluxograma. Em termos de programa, isso significa exercitar uma ou mais condições novas. Ambos os ramos, verdadeiro e falso, de todas as condições devem ser executados.

O fluxograma do procedimento de busca binária é mostrado na Figura 23.14 com cada nó representando uma linha no programa como uma declaração executável. Ao rastrear o fluxo, portanto, você poderá ver que os caminhos no fluxograma de busca binária são:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14
1, 2, 3, 4, 5, 14
1, 2, 3, 4, 5, 6, 7, 11, 12, 5, ...
1, 2, 3, 4, 6, 7, 2, 11, 13, 5, ...

Figura 23.14

Fluxograma para a rotina de busca.



Se todos esses caminhos forem executados, podemos assegurar que cada declaração no método foi executada pelo menos uma vez e que cada ramo foi exercitado quanto às condições verdadeira ou falsa.

Você pode encontrar o número de caminhos independentes em um programa calculando a *complexidade ciclomática* (McCabe, 1976) de um fluxograma de programa. Para programas sem declarações 'goto', o valor da complexidade ciclomática é um a mais do que o número de condições no programa. Uma condição simples é uma expressão lógica sem conectores 'and' ou 'or'. Se o programa incluir condições compostas, que são as expressões lógicas que incluem conectores 'and' ou 'or', você deve contar o número de condições simples nas condições compostas quando se calcula a complexidade ciclomática.

Portanto, se existirem seis declarações *if* e um loop *while*, e todas as expressões condicionais forem simples, a complexidade ciclomática será 8. Se uma expressão condicional for uma expressão composta, como 'if A and B or C', você deve contar isso como três condições simples. A complexidade ciclomática será, portanto, 10. A complexidade ciclomática do algoritmo de busca binária (Figura 23.12) é 4, pois existem três condições simples nas linhas 5, 7 e 11.

Após descobrir o número de caminhos independentes por meio do código pelo cálculo da complexidade ciclomática, você deve, em seguida, projetar os casos de teste para cada um desses caminhos. O número mínimo de casos de teste necessário para testar todos os caminhos é igual à complexidade ciclomática.

O projeto de casos de teste é direto no caso da rotina de busca binária. Contudo, quando o programa tem uma estrutura de ramificação complexa, pode ser difícil prever qual caso de teste específico será processado. Nesses casos, um analisador dinâmico de programa pode ser usado para descobrir o perfil de execução do programa.

Analisadores dinâmicos de programa são ferramentas de teste que trabalham em conjunto com os compiladores. Durante a compilação, esses analisadores adicionam instruções extras ao código gerado. Eles contam o número de vezes em que cada declaração foi executada. Depois que o programa for executado, um perfil de execução poderá ser impresso. Ele mostra que partes do programa foram ou não executadas usando casos de teste específicos. Esse perfil de execução, portanto, revela seções do programa que não foram testadas.

23.4 Automação de testes

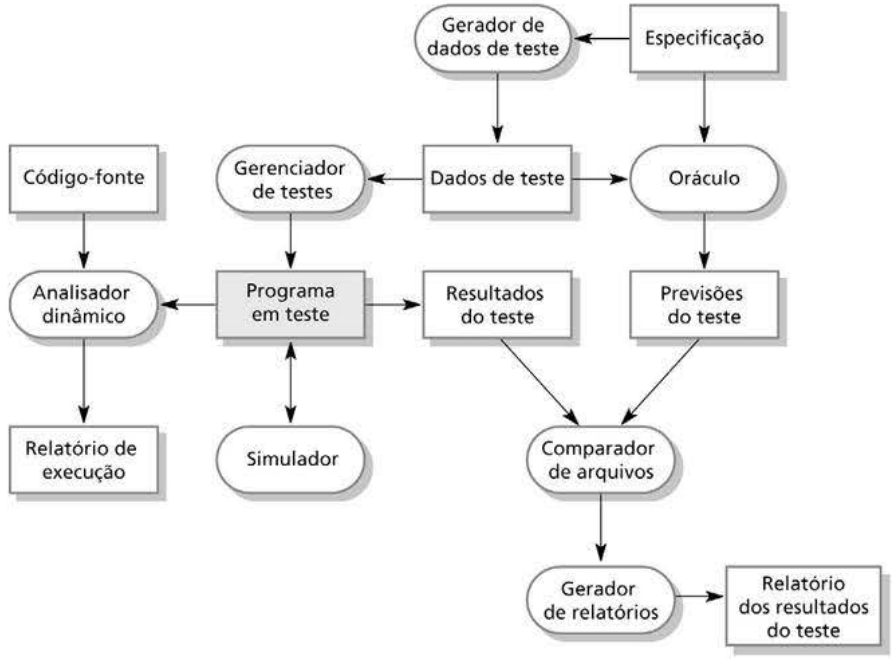
O teste é uma fase dispendiosa e trabalhosa do processo de software. Sendo assim, as ferramentas de teste estão entre as primeiras ferramentas de software a serem desenvolvidas. Atualmente, essas ferramentas oferecem uma variedade de recursos e seu uso pode reduzir significativamente os custos de teste.

Já expliquei uma abordagem para a automação de testes (Mosley e Posey, 2002) na qual um framework de testes, como o JUnit (Massol e Husted, 2003) é usado para teste de regressão. O JUnit é um conjunto de classes em Java que o usuário estende para criar um ambiente de testes automatizado. Cada teste individual é implementado como um objeto e um executor de testes realiza todos os testes. Estes em si devem ser escritos de maneira que indiquem se o sistema testado comportou-se conforme esperado.

Um workbench de testes de software é um conjunto integrado de ferramentas para apoiar o processo de teste. Além dos frameworks de testes que apóiam a execução de teste automatizada, um workbench pode incluir ferramentas para simular outras partes do sistema a fim de gerar dados de teste. A Figura 23.15 mostra algumas das ferramentas que podem ser incluídas nesse workbench de testes:

1. *Gerenciador de testes.* Gerencia a execução de testes de programa. O gerenciador de testes mantém o acompanhamento dos dados de teste, resultados esperados e os recursos de programa testados. Os frameworks de automação de testes, como JUnit, são exemplos de gerenciadores de testes.
2. *Gerador de dados de teste.* Gera os dados de teste para o programa a ser testado. Isso pode ser realizado por meio da seleção de dados de uma banco de dados ou por meio do uso de padrões para gerar dados aleatórios de maneira correta.
3. *Oráculo.* Gera previsões de resultados de teste esperados. Os oráculos podem ser tanto versões anteriores do programa ou protótipos do sistema. O teste *back-to-back* (explicado no Capítulo 17) envolve a execução em paralelo do oráculo e do programa a ser testado. Diferenças entre as saídas são evidenciadas.
4. *Comparador de arquivos.* Compara os resultados dos testes de programa com os resultados de testes anteriores e relata as diferenças entre elas. Os comparadores são usados em testes de regressão nos quais são comparados os resultados da execução de diferentes versões. Quando os testes automatizados são usados, eles podem ser chamados de dentro dos próprios testes.
5. *Gerador de relatórios.* Fornece recursos de definição e de geração de relatórios para os resultados de teste.
6. *Analisador dinâmico.* Adiciona código ao programa para contar o número de vezes em que cada declaração foi executada. Depois do teste, um perfil de execução é gerado para mostrar a freqüência com que cada declaração de programa foi executada.

Figura 23.15
Workbench de testes.



7. *Simulador.* Diferentes tipos de simuladores podem ser fornecidos. Simuladores de alvo simulam a máquina na qual o programa vai ser executado. Simuladores de interface com o usuário são programas dirigidos a scripts que simulam várias interações simultâneas de usuários. O uso de simuladores de E/S significa que o timing de seqüências de transações é repetível.

Quando usados para teste de sistemas de grande porte, as ferramentas devem ser configuradas e adaptadas para o sistema específico a ser testado. Por exemplo:

1. Novas ferramentas podem ser adicionadas para testar características específicas da aplicação e algumas ferramentas de teste existentes podem não ser necessárias.
2. Podem ser escritos scripts para simuladores de interface com o usuário e padrões definidos para os geradores de dados de teste. Formatos de relatório podem também ser definidos.
3. Conjuntos de resultados esperados de teste podem ser preparados manualmente caso as versões anteriores do programa não estejam disponíveis para funcionarem como um oráculo.
4. Podem ser escritos comparadores de arquivos com propósitos específicos e que incluam o conhecimento da estrutura dos resultados do teste no arquivo.

Uma quantidade significativa de esforço e tempo geralmente é necessária para criar um workbench de teste abrangente. Workbenches de testes completos, conforme mostrados na Figura 23.15, portanto, são usados somente quando sistemas de grande porte estão em desenvolvimento. Para esses sistemas, os custos de teste podem atingir 50% dos custos totais de desenvolvimento, de modo que é adequado em termos de custo investir em ferramentas CASE de alta qualidade para apoiar os testes. Contudo, devido aos diferentes tipos de sistemas exigirem diferentes tipos de apoio ao teste, ferramentas de teste comerciais podem não estar disponíveis. Rankin (Rankin, 2002) explica uma situação na IBM e descreve o projeto de um sistema de apoio ao teste desenvolvido para um servidor de e-business.

PONTOS-CHAVE

- Os testes podem mostrar somente a presença de erros em um programa. Eles não podem demonstrar que não existam defeitos remanescentes.
- O teste de componentes é de responsabilidade do desenvolvedor de componentes. Outra equipe de testes geralmente executa os testes do sistema.
- O teste de integração é a atividade inicial de teste de sistema, na qual são testados os componentes integrados para detectar defeitos. O teste de releases concentra-se em testar releases do cliente e em validar se o sistema a ser liberado atende aos requisitos.
- Ao testar sistemas, você deve tentar 'quebrar' o sistema usando sua experiência e diretrizes para escolher os tipos de casos de teste eficazes na descoberta de defeitos em outros sistemas.
- O teste de interfaces tenta descobrir defeitos nas interfaces dos componentes compostos. Defeitos de interface podem surgir devido a erros de leitura da especificação, má interpretação da especificação ou erros ou suposições inválidas de timing.
- O particionamento de equivalência é uma maneira de derivar casos de teste. Ele depende da descoberta de partições nos conjuntos de dados de entrada e de saída e do exercício do programa com valores dessas partições. Freqüentemente, o valor mais provável de conduzir a um teste bem-sucedido é um valor no limite da partição.
- O teste estrutural baseia-se na análise de um programa para determinar seus caminhos e no uso dessa análise para ajudar na seleção dos casos de teste.
- A automação reduz os custos de teste pelo apoio ao processo de teste com uma variedade de ferramentas de software.

LEITURAS SUGERIDAS

How to break software: a practical guide to testing. Este é um livro mais prático do que teórico sobre testes de software, no qual o autor apresenta um conjunto de diretrizes baseadas em experiências com o projeto de testes, as quais são eficazes na descoberta de defeitos de sistemas. (J. A. Whittaker, 2002, Addison-Wesley.)

"Software testing and verification". Esta publicação especial do *IBM Systems Journal* inclui uma série de artigos sobre teste, entre eles uma boa visão geral, artigos sobre métricas de teste e automação de testes (*IBM Systems Journal*, 41(1), janeiro de 2002).

Testing object-oriented systems: models, patterns and tools. Este imenso livro fornece uma cobertura completa sobre testes orientados a objetos. Pela extensão, fica claro que esse não deve ser o primeiro livro a ser lido sobre o assunto (a maioria dos livros sobre

orientação a objetos tem um capítulo de teste), mas é de fato um livro fundamental sobre testes orientado a objetos. (R. V. Binder, 1999, Addison-Wesley.)

"How to design practical test cases". Um artigo do tipo 'como fazer' sobre projeto de casos de teste de um autor de uma empresa japonesa com uma reputação muito boa no que se refere à entrega de software com muito poucos defeitos. (T. Yamaura, *IEEE software*, 15(6), novembro de 1998.)

EXERCÍCIOS

23.1 Explique por que os testes podem somente detectar a presença de erros, e não sua ausência.

23.2 Compare a integração e teste top-down e bottom-up por meio da explicação de suas vantagens e desvantagens em relação ao teste de arquitetura, à demonstração de uma versão do sistema aos usuários e à implementação prática e observação de testes. Explique por que a integração de sistemas de grande porte, na prática, precisa usar uma combinação das abordagens top-down e bottom-up.

23.3 O que é teste de regressão? Explique como o uso de testes automatizados e framework de testes, como o JUnit, simplifica o teste de regressão.

23.4 Escreva um cenário que possa ser usado como base para a derivação de testes para o sistema de estação meteorológica usado como exemplo no Capítulo 14.

23.5 Usando o diagrama de sequência da Figura 8.12 como cenário, proponha testes para a produção de itens eletrônicos no sistema LIBSYS.

23.6 Quais são os problemas no desenvolvimento de testes de desempenho para um sistema de banco de dados distribuídos, como o sistema LIBSYS?

23.7 Explique por que é necessário o teste de interfaces mesmo quando componentes individuais foram extensivamente validados por meio de testes de componentes e inspeções de programa.

23.8 Usando a abordagem explicada aqui para teste de objetos, projete casos de teste para testar os estados do forno de microondas cujo modelo de estados está definido na Figura 8.5.

23.9 Foi solicitado que você teste um método chamado **catWhiteSpace** em um objeto **Paragraph** que, dentro de um parágrafo, substitui seqüências de caracteres em branco por um único caracter em branco. Identifique as partições de teste desse exemplo e derive um conjunto de testes para o método **catWhiteSpace**.

23.10 Forneça três situações em que o teste de todos os caminhos independentes de um programa possa não detectar erros de programa.

24

Validação de sistemas críticos

Objetivos

O objetivo deste capítulo é explicar técnicas de verificação e validação usadas no desenvolvimento de sistemas críticos. Após ler este capítulo, você:

- compreenderá como a confiabilidade de um sistema de software pode ser medida e como os modelos de crescimento de confiabilidade podem ser usados para prever quando determinado nível de confiabilidade será atingido;
- compreenderá os princípios de argumentos de segurança e como eles podem ser usados junto com outros métodos de V & V para garantir a segurança do sistema;
- compreenderá os problemas de assegurar a proteção de um sistema;
- terá sido apresentado a casos de segurança com argumentos e evidências de segurança de sistema.

Conteúdo

24.1 Validação de confiabilidade
24.2 Garantia de segurança
24.3 Avaliação de proteção
24.4 Casos de segurança e confiabilidade

A verificação e validação de um sistema crítico têm, obviamente, muito em comum com a validação de qualquer outro sistema. Os processos de V & V devem demonstrar que o sistema atende às especificações e que os serviços e comportamento do sistema apóiam os requisitos do cliente. Contudo, para sistemas críticos, nos quais um alto grau de confiabilidade é exigido, testes e análises adicionais são necessários para produzir evidências de que o sistema é confiável. Existem duas razões pelas quais você deve fazer isso:

1. *Custos das falhas*. Os custos e as conseqüências de falha de sistemas críticos são potencialmente muito maiores do que em sistemas não críticos. Os riscos de falha de sistema são diminuídos por meio de investimento adicional em verificação e validação do sistema. É geralmente mais barato descobrir e remover defeitos antes que o sistema seja entregue do que pagar pelos custos conseqüentes de acidentes ou interrupções de serviços do sistema.
2. *Validação de atributos de confiabilidade*. Pode ser necessário desenvolver um caso formal para os clientes demonstrando que o sistema atende aos requisitos de confiabilidade

especificados (disponibilidade, confiabilidade, segurança e proteção). A avaliação dessas características de confiabilidade requer atividades específicas de V & V explicadas mais adiante neste capítulo. Em alguns casos, os agentes regulamentadores externos, como autoridades nacionais de aviação, podem precisar se certificar de que o sistema é seguro antes que ele seja entregue. Para obter essa certificação, pode ser necessário projetar e realizar procedimentos específicos de V & V que coletem evidências sobre a confiabilidade do sistema.

Por essas razões, os custos de V & V em sistemas críticos são geralmente muito maiores do que em outras classes de sistemas. É normal que V & V consuma mais de 50% do total de custos de desenvolvimento de sistemas de software críticos. Esse custo é, naturalmente, justificável por evitar uma dispendiosa falha do sistema. Por exemplo, em 1996, um sistema de software crítico da missão do foguete Ariane 5 falhou e vários satélites foram destruídos. As perdas foram de centenas de milhões de dólares. A investigação posterior descobriu que deficiências no V & V do sistema foram parcialmente responsáveis por essa falha.

Embora o processo de validação de sistemas críticos deva principalmente enfocar a validação do sistema, atividades relacionadas devem verificar se os processos de desenvolvimento de sistema definidos foram seguidos. Conforme explico nos capítulos 27 e 28, a qualidade do sistema é afetada pela qualidade dos processos usados para desenvolvê-lo. Em resumo, bons processos levam a bons sistemas. Portanto, para produzir sistemas confiáveis, você precisa estar convicto de que um processo confiável de desenvolvimento foi seguido.

Essa garantia de processo é parte inerente dos padrões ISO 9000 do gerenciamento de qualidade, explicado brevemente no Capítulo 27. Esses padrões requerem documentação dos processos usados e atividades associadas que garantam que o processo foi seguido. Isso requer a geração de registros de processo, como formulários com assinaturas, que certifiquem a realização de atividades de processo e verificações de qualidade de produto. Os padrões ISO 9000 especificam que devem ser produzidas saídas de processo tangíveis e quem é responsável por produzi-las. Na Seção 24.2.2, apresento um exemplo de registro de processo de análise de perigos.

24.1 Validação de confiabilidade

Conforme expliquei no Capítulo 9, uma série de métricas foram desenvolvidas para especificar os requisitos de confiabilidade de um sistema. Para validar se o sistema atende a esses requisitos, você precisa medir a confiabilidade do sistema quando visto por um usuário típico.

O processo de medição de confiabilidade de um sistema é ilustrado na Figura 24.1. Esse processo envolve quatro estágios:

1. Você começa pelo estudo de sistemas existentes do mesmo tipo para estabelecer um perfil operacional. Um perfil operacional identifica categorias de entradas e a probabilidade de que essas entradas ocorram em uso normal.
2. Você então constrói um conjunto de dados de teste que reflitam o perfil operacional. Isso significa que são criados dados de teste com a mesma distribuição de probabilidade que os dados de teste para o sistema que você estudou. Normalmente, é usado um gerador de dados de teste para apoiar esse processo.
3. Você testa o sistema usando esses dados e conta o número e tipo de falhas que ocorrem. Os tempos das falhas também devem ser registrados. Conforme explicado no Capítulo 9, as unidades de tempo escolhidas devem ser apropriadas à métrica de confiabilidade usada.
4. Depois de observar o número de falhas estatisticamente significativas, você pode calcular a confiabilidade do software e desenvolver o valor da métrica de confiabilidade apropriado.

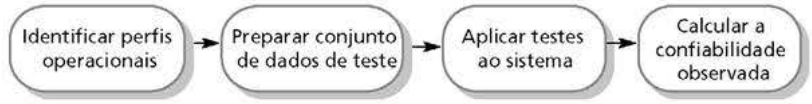
Essa abordagem é muitas vezes chamada de *teste estatístico*. O objetivo do teste estatístico é avaliar a confiabilidade do sistema. Ela contrasta com o teste de defeitos, explicado no Capítulo 23, no qual o objetivo é descobrir defeitos do sistema. Prowell et al. (Prowell, et al., 1999) fornecem uma boa descrição desses testes estatísticos em seu livro sobre engenharia de software Cleanroom.

Essa abordagem conceitualmente atraente para medição de confiabilidade não é fácil de ser aplicada na prática. As principais dificuldades que emergem são devidas a:

1. *Incerteza de perfil operacional*. Os perfis operacionais baseados na experiência com outros sistemas podem não ser um reflexo preciso do uso real do sistema.

Figura 24.1

Processo de medição de confiabilidade.



2. *Altos custos de geração de dados de teste*. Pode ser muito dispendioso gerar o grande volume de dados necessários em um perfil operacional a menos que o processo seja totalmente automatizado.
3. *Incerteza estatística quando uma alta confiabilidade é especificada*. Você deve gerar um número de falhas estatisticamente significativas para permitir medições de confiabilidade precisas. Quando o software já é confiável, ocorrem relativamente poucas falhas e é difícil gerar novas falhas.

O desenvolvimento de um perfil operacional preciso é certamente possível para alguns tipos de sistemas, como sistemas de telecomunicações, que têm um modelo de uso padronizado. Para outros tipos de sistema, contudo, existem muitos usuários diferentes sendo que cada um tem seu próprio modo de usar o sistema. Conforme expliquei no Capítulo 3, diferentes usuários podem ter impressões diferentes sobre a confiabilidade, pois usam o sistema de maneiras diferentes.

Sem dúvida alguma, a melhor maneira de gerar o grande conjunto de dados necessário para a medição de confiabilidade é usar um gerador de dados de teste que possa ser configurado para gerar automaticamente entradas que combinem com o perfil operacional. Entretanto, geralmente não é possível automatizar a produção de todos os tipos de dados de teste para sistemas interativos, pois as entradas freqüentemente são uma resposta às saídas do sistema. Os conjuntos de dados desses sistemas precisam ser gerados manualmente, com custos correspondentes maiores. Mesmo quando a automação completa é possível, escrever comandos para o gerador de dados de teste pode exigir um tempo significativo.

A incerteza estatística é um problema geral na medição de confiabilidade de um sistema. Para fazer uma previsão precisa de confiabilidade, é necessário mais do que simplesmente causar uma única falha no sistema. Você precisa gerar uma quantidade razoavelmente grande, estatisticamente significativa, para estar convicto de que sua medição de confiabilidade é precisa. Assim, quanto mais você minimiza o número de defeitos em um sistema, mais difícil se torna medir a eficiência das técnicas de minimização de defeitos. Se forem especificados níveis muito altos de confiabilidade, se tornará impraticável gerar uma quantidade suficiente de falhas no sistema para verificar essas especificações.

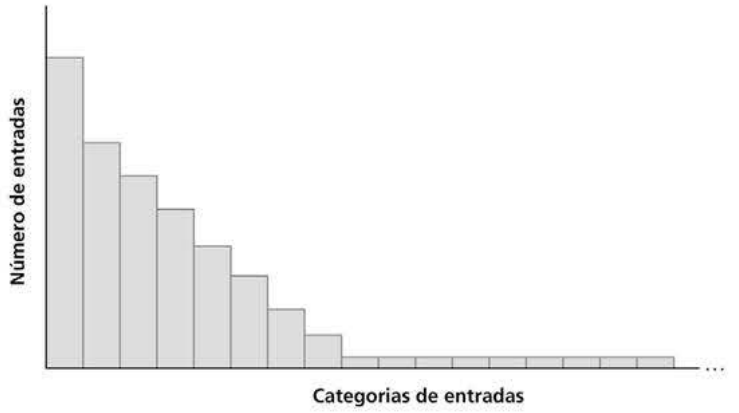
24.1.1 Perfis operacionais

O perfil operacional de um software reflete como ele será usado na prática. Ele consiste de uma especificação de categorias de entradas e a probabilidade de sua ocorrência. Quando um novo software substitui um sistema manual ou automatizado, é razoavelmente fácil avaliar o provável padrão de uso do novo software. Ele deve corresponder ao uso atual, com alguma concessão feita à nova funcionalidade que está (presumivelmente) incluída no novo software. Por exemplo, pode ser especificado um perfil operacional para sistemas de comutação da área de telecomunicações, pois as empresas de telecomunicações conhecem os padrões de chamadas com os quais esses sistemas precisam lidar.

Tipicamente, o perfil operacional é organizado de modo que as entradas com maiores probabilidades de serem geradas classificam-se em um pequeno número de categorias, conforme mostrado à esquerda da Figura 24.2. Existe um número extremamente grande de categorias em que as entradas são altamente improváveis, mas não impossíveis. Estes são mostrados à direita da Figura 24.2. As reticências (...) significam que existem muito mais dessas entradas não usuais do que as mostradas.

Musa (Musa, 1993; Musa, 1998) sugere diretrizes para o desenvolvimento de perfis operacionais. Ele trabalhou com engenharia de sistemas de telecomunicações e existe uma longa história de coleta de dados de uso nesse domínio. Conseqüentemente, o processo de desenvolvimento de perfil operacional é relativamente direto. Para um sistema que requer aproximadamente 15 pessoas-ano de esforço de desenvolvimento, foi desenvolvido um perfil operacional com aproximadamente 1 pessoa-mês de esforço. Em outros casos, a geração do perfil operacional levou muito tempo (2 a 3 pessoas-ano), mas o custo foi distribuído em uma série de releases de sistema. Musa reconheceu que sua empresa (uma empresa de telecomunicações) tinha um retorno de investimento de pelo menos 10:1 necessário para desenvolver um perfil operacional.

Figura 24.2
Perfil operacional.



Contudo, quando um sistema de software é novo e pioneiro, é difícil antecipar como ele será usado e, conseqüentemente, gerar um perfil operacional preciso. Muitos usuários diferentes com expectativas, conhecimentos e experiência diferentes podem usar o novo sistema. Não existe um banco de dados com histórico de uso. Esses usuários podem fazer uso de sistemas de maneiras não previstas pelos desenvolvedores do sistema.

O problema é mais complexo, pois os perfis operacionais podem mudar à medida que o sistema é usado. Conforme os usuários aprendem sobre um novo sistema e tornam-se mais confiantes, eles usam o sistema de maneira mais sofisticada. Por conta dessas dificuldades, Hamlet (Hamlet, 1992) sugere muitas vezes que é impossível desenvolver um perfil operacional confiável. Se você não está certo de que seu perfil operacional está correto, você não pode estar convicto sobre a precisão de suas medições de confiabilidade.

24.1.2 Previsão de confiabilidade

Durante a validação de software, os gerentes devem se dedicar ao teste de sistema. Como o teste é muito dispendioso, é importante interrompê-lo tão logo seja possível, sem testar o sistema além do necessário. O teste poderá ser interrompido quando o nível necessário de confiabilidade do sistema for atingido. Algumas vezes, naturalmente, as previsões de confiabilidade podem revelar que o nível necessário de confiabilidade nunca será atingido. Nesse caso, o gerente deve tomar decisões difíceis quanto a reescrever partes do software ou renegociar o contrato do sistema.

O modelo de crescimento de confiabilidade é um modelo de como a confiabilidade do sistema muda no período do processo de teste. À medida que as falhas são descobertas, os defeitos subjacentes que causam essas falhas são reparados de maneira que a confiabilidade do sistema deve aumentar durante o teste e o debugging do sistema. Para prever a confiabilidade, o modelo conceitual de crescimento de confiabilidade deve então ser traduzido em um modelo matemático. Não abordo esse nível de detalhes nesta seção, simplesmente explico o princípio de aumento da confiabilidade.

Existem vários modelos de aumento de confiabilidade derivados de experimentos em uma série de diferentes domínios de aplicação. Segundo Kan (Kan, 2003), a maioria desses modelos é exponencial, cuja confiabilidade aumenta à medida que os defeitos são descobertos e removidos (veja a Figura 24.5). O aumento então cessa progressivamente e atinge um patamar em que cada vez menos defeitos são descobertos e removidos nas fases finais do teste.

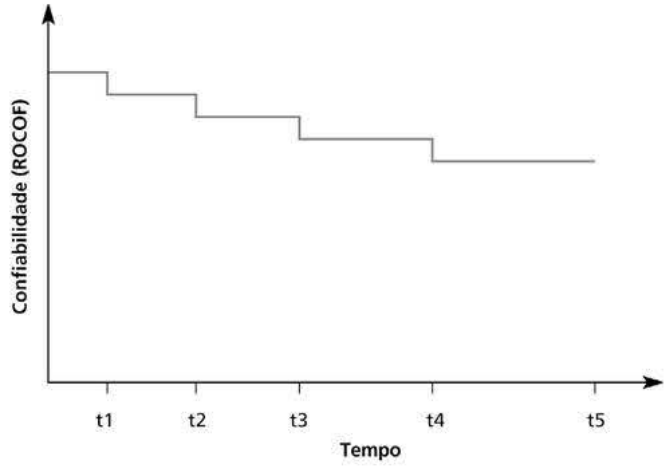
O modelo mais simples que ilustra o conceito de aumento de confiabilidade é o modelo em etapas iguais (Jelinski e Moranda, 1972). A confiabilidade aumenta em incrementos constantes cada vez que um defeito (ou um conjunto de defeitos) é descoberto e corrigido (Figura 24.3) e uma nova versão do software é criada. Esse modelo supõe que as correções de software são sempre adequadamente implementadas, de modo que o número de defeitos de software e as falhas associadas diminuem a cada nova versão do sistema. À medida que os reparos são realizados, a taxa de ocorrência de falhas de software (ROCOF — *Rate of Occurrence of Software Failures*) deve, portanto, diminuir, conforme mostrado na Figura 24.3. Observe que os períodos de tempo no eixo horizontal refletem o tempo entre releases do sistema por teste; assim, eles normalmente apresentam durações diferentes.

Na prática, contudo, os defeitos de software nem sempre são corrigidos durante o debugging e, quando se muda um programa, algumas vezes são introduzidos novos defeitos. A probabilidade de ocorrência desses defeitos pode ser maior do que a probabilidade de ocorrência do defeito corrigido. Portanto, a confiabilidade do sistema pode algumas vezes piorar em um novo release em vez de melhorar.

O simples modelo de aumento da confiabilidade em etapas iguais também pressupõe que todos os defeitos contribuem igualmente para a confiabilidade e que cada correção de defeito contribui com a mesma quantidade de aumento da con-

Figura 24.3

Modelo de aumento de confiabilidade em etapas iguais.



fiabilidade. Contudo, nem todos os defeitos são igualmente prováveis. A correção da maioria dos defeitos mais comuns contribui mais para o aumento da confiabilidade do que para a correção de defeitos ocasionais. Você pode localizar esses prováveis defeitos com antecedência no processo de teste e, por isso, a confiabilidade pode aumentar mais do que quando, posteriormente, os defeitos menos prováveis forem descobertos.

Modelos posteriores, como o sugerido por Littlewood e Verral (Littlewood e Verral, 1973), levam em conta esses problemas por meio da introdução de um elemento aleatório no aumento da confiabilidade obtido por uma correção de software. Assim, cada correção não resulta em uma quantidade igual de aumento da confiabilidade, mas varia dependendo do elemento aleatório (Figura 24.4).

O modelo de Littlewood e Verral permite o aumento negativo de confiabilidade quando uma correção de software introduz erros adicionais. Ele também modela o fato de que, quando os defeitos são corrigidos, o aumento médio na confiabilidade por correção diminui. A razão disso é que os defeitos mais prováveis podem ser descobertos antecipadamente no processo de teste. A correção desses defeitos contribui mais para o aumento da confiabilidade.

Os modelos citados antes são modelos discretos que refletem o aumento incremental da confiabilidade. Quando uma nova versão do software com defeitos corrigidos é liberada para teste, ela deve ter uma taxa de ocorrência de falhas menor do que a versão anterior. Contudo, para prever a confiabilidade atingida depois de determinada quantidade de testes, são necessários modelos matemáticos contínuos. Muitos modelos, derivados de diferentes domínios de aplicação, foram propostos e comparados (Littlewood, 1990).

De maneira simples, você pode prever a confiabilidade por meio da combinação de dados medida com um modelo de confiabilidade conhecido. Você pode, depois, incrementar o modelo para o nível de confiabilidade necessário e observar quando esse nível será atingido (Figura 24.5). Portanto, o teste e o debugging devem continuar até esse momento.

Figura 24.4

Modelo de aumento de confiabilidade em etapas aleatórias.

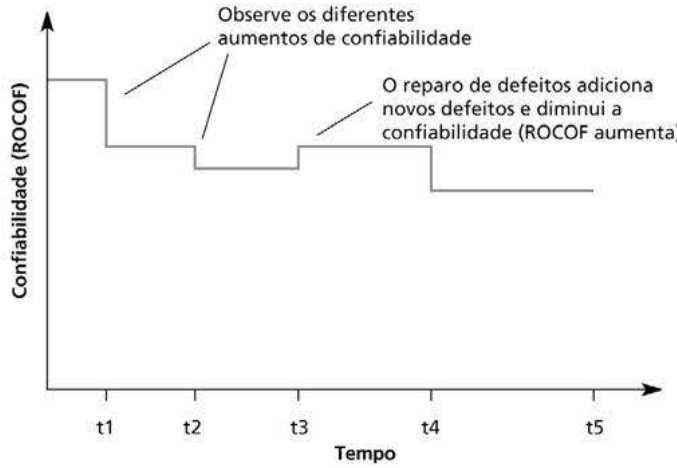
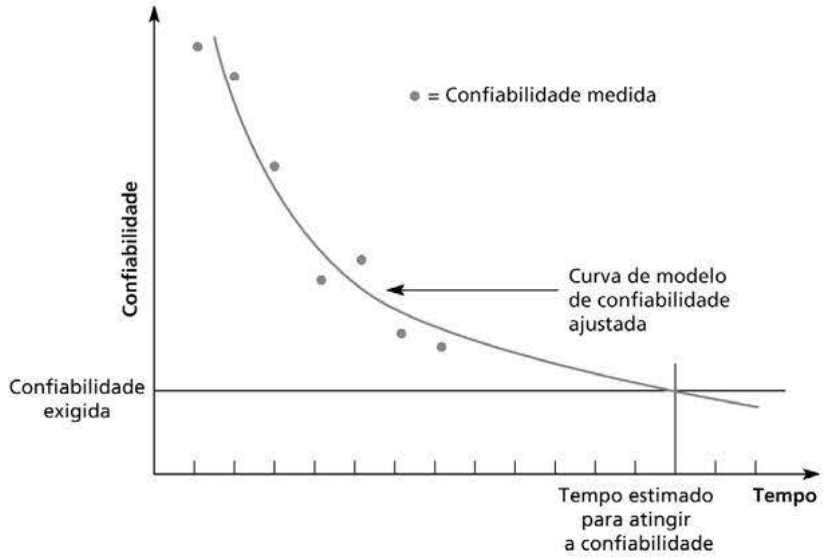


Figura 24.5

Previsão de confiabilidade.



A previsão de confiabilidade do sistema com base em um modelo de aumento da confiabilidade tem dois benefícios principais:

1. *Planejamento de testes*. Dado o cronograma de testes atual, você pode prever quando o teste estará completo. Se isso ocorrer depois da data de entrega planejada do sistema, você poderá precisar implantar recursos adicionais para teste e debugging para acelerar a taxa de aumento de confiabilidade.
2. *Negociações com o cliente*. Algumas vezes, o modelo de confiabilidade mostra que o crescimento da confiabilidade é muito lento e que uma quantidade desproporcional de esforço de teste é necessária para obter, relativamente, pouco benefício. Pode valer a pena renegociar os requisitos de confiabilidade com o cliente. Pode ser também que o modelo preveja que a confiabilidade necessária provavelmente nunca será atingida. Nesse caso, você terá que renegociar os requisitos de confiabilidade do sistema com o cliente.

Simplifiquei a modelagem de aumento da confiabilidade nesta seção para que você compreenda o conceito. Se desejar usar esses modelos, você deverá se aprofundar e compreender a matemática presente nesses modelos e seus problemas práticos. Littlewood e Musa (Littlewood, 1990; Abdel-Ghaly, et al., 1986; Musa, 1998) escreveram extensivamente sobre modelos de aumento da confiabilidade, e Kan (Kan, 2003) apresenta um excelente resumo em seu livro. Vários autores descreveram suas experiências práticas no uso de modelos de aumento da confiabilidade (Ehrlich, et al., 1993; Schneidewind e Keller, 1992; Sheldon, et al., 1992).

24.2 Garantia de segurança

Os processos de garantia de segurança e validação de confiabilidade têm objetivos diferentes. Você pode especificar a confiabilidade quantitativamente usando alguma métrica e, depois, medir a confiabilidade do sistema finalizado. Dentro dos limites do processo de medição, você sabe se o nível necessário de confiabilidade foi atingido. A segurança, contudo, não pode ser significativamente especificada de maneira quantitativa e não pode ser medida quando o sistema é testado.

A garantia de segurança concentra-se, portanto, em definir um nível de confiabilidade do sistema que pode variar de 'muito baixo' até 'muito alto'. Esse é um assunto para um julgamento profissional baseado em um conjunto de evidências sobre o sistema, seu ambiente e seu processo de desenvolvimento. Em muitos casos, essa confiabilidade baseia-se, em parte, na experiência da organização que desenvolve o sistema. Se uma empresa desenvolveu previamente uma série de sistemas de controle que operaram de maneira segura, é razoável presumir que ela continuará a desenvolver sistemas seguros desse tipo.

Contudo, tal avaliação deve ser sustentada por evidências tangíveis com base no projeto de sistema, dos resultados de V & V do sistema e dos processos de desenvolvimento de sistema usados. Para alguns sistemas, essas evidências tangíveis são montadas em um caso de segurança (veja a Seção 24.4) que permite que um agente regulamentador externo chegue à conclusão de que a confiança do desenvolvedor na segurança do sistema é justificada.

Os processos de V & V para sistemas críticos de segurança têm muito em comum com os processos comparáveis de quaisquer outros sistemas com altos requisitos de confiabilidade. Deve haver testes extensivos para descobrir tantos defeitos quanto possíveis e, quando apropriado, testes estatísticos podem ser usados para avaliar a confiabilidade do sistema. Contudo, devido à baixíssima taxa de falhas exigida em muitos sistemas críticos de segurança, os testes estatísticos nem sempre podem fornecer uma estimativa quantitativa da confiabilidade do sistema. Os testes fornecem simplesmente alguma evidência, que é usada junto com outras evidências, como resultados de revisões e verificações estáticas (veja o Capítulo 22), para fazer um julgamento sobre a segurança do sistema.

Revisões extensivas são essenciais durante o processo de desenvolvimento orientado à segurança para expor o software a pessoas que irão percebê-lo de diferentes perspectivas. Parnas et al. (Parnas, et al., 1990) sugerem cinco tipos de revisões que devem ser obrigatórios para sistemas críticos de segurança:

1. revisão de função pretendida correta;
2. revisão de estrutura fácil de manter e compreender;
3. revisão para verificar se o algoritmo e o projeto de estrutura de dados são consistentes com o comportamento especificado;
4. revisão de consistência do código, do algoritmo e do projeto de estrutura de dados;
5. revisão de adequação dos casos de teste de sistema.

Uma assertiva que fundamenta o trabalho em segurança de sistema é que o número de defeitos do sistema, que pode levar a perigos críticos de segurança, é significativamente menor do que o número total de defeitos que podem existir no sistema. A garantia de segurança pode se concentrar nesses defeitos de perigo potencial. Se for possível demonstrar que

esses defeitos não podem ocorrer ou, mesmo que ocorram, o perigo associado não resultará em acidente, o sistema é seguro. Essa é a base de argumentos de segurança explicados na próxima seção.

24.2.1 Argumentos de segurança

As provas de correção de programas, conforme explicado no Capítulo 22, foram propostas como uma técnica de verificação de software há mais de 30 anos. Provas formais de programa podem certamente ser criadas para sistemas pequenos. Contudo, as dificuldades práticas para provar que um sistema atende às suas especificações são tão grandes que poucas organizações consideram as provas de correção adequadas em termos de custo. Contudo, para algumas aplicações críticas, pode compensar desenvolver provas de correção para aumentar a confiança de que o sistema atende aos requisitos de segurança ou de proteção. Esse é particularmente o caso quando a funcionalidade crítica de segurança pode ser isolada em um subsistema bastante pequeno que pode ser formalmente especificado.

Embora possa não ser adequado em termos de custo desenvolver provas de correção para a maioria dos sistemas, algumas vezes é possível desenvolver argumentos de segurança mais simples que demonstram que o programa atende a suas obrigações de segurança. Em um argumento de segurança, não é necessário provar que a funcionalidade do sistema está de acordo com a especificação. É necessário somente demonstrar que a execução do programa não pode resultar em um estado inseguro.

A técnica mais eficaz para demonstrar a segurança de um sistema é a prova por contradição. Começa-se supondo que um estado inseguro, identificado pela análise de perigos, pode ser atingido pela execução do programa. É escrito um predicado que define esse estado inseguro. O código é, portanto, analisado sistematicamente, mostrando que, por todos os caminhos de programa que conduzem àquele estado, a condição de término desses caminhos contradiz o predicado de estado inseguro. Se esse for o caso, a hipótese inicial de estado inseguro é incorreta. Se isso se repetir para todos os perigos identificados, o software é seguro.

Como exemplo, considere o código da Figura 24.6, que poderia ser parte da implementação do sistema de fornecimento de insulina. O desenvolvimento de um argumento de segurança para esse código envolve demonstrar que a dose de insulina administrada nunca é maior do que o nível máximo definido para cada pessoa diabética. Portanto, não é necessário provar que o sistema fornece a dose correta, mas apenas que nunca fornece uma dose excessiva ao paciente.

Para criar o argumento de segurança, você deve identificar a precondição para o estado inseguro que, nesse caso, é **currentDose > maxDose**. Você, então, deve demonstrar que todos os caminhos de programa conduzem à contradição dessa asserção de insegurança. Se esse for o caso, a condição insegura pode não ser verdadeira. Portanto, o sistema é seguro. Você pode estruturar e apresentar os argumentos de segurança graficamente, conforme mostrado na Figura 24.7.

Argumentos de segurança, como os mostrados na Figura 24.7, são muito mais curtos do que verificações formais de sistema. Primeiro, você deve identificar todos os caminhos possíveis que conduzem potencialmente a um estado inseguro. O trabalho é feito para trás a partir desse estado inseguro, considerando a última atribuição para todas as variáveis de estado em cada caminho que conduz a ele. Você pode ignorar os cálculos prévios (como a declaração if 1, na Figura 24.7) no

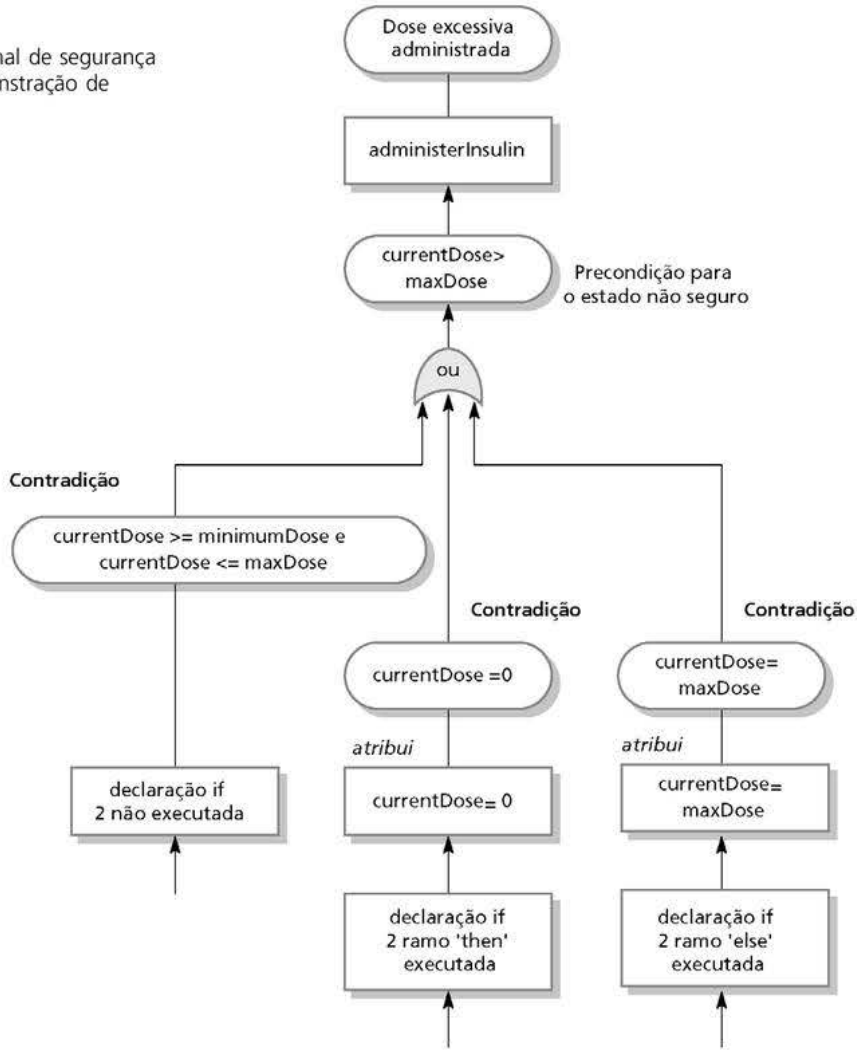
Figura 24.6

Programa de abastecimento de insulina

```
– a dose de insulina a ser fornecida é uma função do
– nível de açúcar no sangue, da dose anterior fornecida e
– tempo decorrido desde o fornecimento da dose anterior

    currentDose = computeInsulin () ;

    // Verificação de segurança–ajustar currentDose, se necessário

    // declaração if 1

    if (previousDose == 0)
    {
        if (currentDose > 16)
                currentDose = 16 ;
    }
    else
        if (currentDose > (previousDose * 2) )
                currentDose = previousDose * 2 ;

    // declaração if 2

    if ( currentDose < minimumDose )
                currentDose = 0 ;
    else if ( currentDose > maxDose )
                currentDose = maxDose ;
    administerInsulin (currentDose) ;
```

Figura 24.7

Argumento informal de segurança baseado na demonstração de contradições.



argumento de segurança. Nesse exemplo, tudo de que você necessita está relacionado ao conjunto de possíveis valores de **currentDose**, imediatamente antes que o método **administerInsulin** seja executado.

No argumento de segurança mostrado na Figura 24.7, existem três possíveis caminhos de programa que conduzem à chamada do método **administerInsulin**. Desejamos demonstrar que a quantidade de insulina fornecida nunca excede **maxDose**. Todos os possíveis caminhos para o método **administerInsulin** são considerados:

Nenhum ramo da declaração if 2 é executado. Isso somente poderá ocorrer se **currentDose** for maior que ou igual a **minimumDose** e menor do que ou igual a **maxDose**.

2. O ramo 'then' da declaração if 2 é executado. Nesse caso, a atribuição de **currentDose** como zero é executada. Portanto, sua pós-condição é **currentDose = 0**.
3. O ramo 'else' da declaração if 2 é executado. Nesse caso, a atribuição de **currentDose** como **maxDose** é executada. Portanto, sua pós-condição é **currentDose = maxDose**.

Em todos os três casos, as pós-condições contradizem a precondição de insegurança de que a dose administrada é maior do que **maxDose**; portanto, o sistema é seguro.

24.2.2 Garantia de processo

Já expliquei a importância de assegurar a qualidade do processo de desenvolvimento de sistema na introdução deste capítulo. Isso é importante para todos os sistemas críticos, mas é particularmente importante para sistemas críticos de segurança. Existem duas razões para isso:

1. Acidentes são eventos raros em sistemas críticos e pode ser praticamente impossível simulá-los durante o teste de um sistema. Você não pode contar com testes extensivos para reproduzir as condições que podem levar a um acidente.
2. Os requisitos de segurança, conforme explicado no Capítulo 9, são algumas vezes requisitos do tipo 'não deve' que excluem comportamentos inseguros do sistema. É impossível demonstrar conclusivamente por testes e outras atividades de validação que esses requisitos foram atendidos.

O modelo de ciclo de vida para o desenvolvimento de sistemas críticos de segurança (Capítulo 9, Figura 9.7) torna claro que deve ser dada atenção explícita à segurança em todos os estágios do processo de software. Isso significa que atividades específicas de garantia de segurança devem ser incluídas no processo. Essas atividades incluem:

- A criação de um sistema de *log* e monitoração de perigos que rastreie os perigos com base na análise preliminar até o teste e validação do sistema.
- A designação de engenheiros de segurança de projeto com responsabilidade explícita pelos aspectos de segurança do sistema.
- O uso extensivo de revisões de segurança durante o processo de desenvolvimento.
- A criação do sistema de certificação de segurança pela qual a segurança de componentes críticos seja formalmente certificada.
- O uso de um sistema de gerenciamento de configuração muito detalhado (veja o Capítulo 29), usado para rastrear toda documentação relacionada à segurança e mantê-la atualizada com a documentação técnica associada. Há pouca relevância em ter procedimentos de validação rigorosos se uma falha de gerenciamento de configuração significa que o sistema errado é entregue ao cliente.

Para ilustrar a garantia de segurança, uso o processo de análise de perigos que é parte essencial do desenvolvimento de sistemas críticos em segurança. A análise de perigos concentra-se em identificar perigos, sua probabilidade e a probabilidade de que esses perigos levem a um acidente. Se o processo de desenvolvimento inclui rastreabilidade clara baseada na identificação de perigos até o sistema em si, então um argumento pode ser definido, pois esses perigos não resultarão em acidentes. Isso pode ser complementado por argumentos de segurança, conforme explicado na Seção 24.2.1. Quando é necessário certificação externa antes que um sistema seja usado (por exemplo, em uma aeronave), geralmente é uma condição da certificação que essa rastreabilidade seja demonstrada.

O documento principal de segurança é o log de perigo, no qual os perigos identificados durante o processo de especificação são documentados e rastreados. Esse log de perigo é, portanto, usado em cada estágio do processo de desenvolvimento de software para avaliar como esses estágios de desenvolvimento levaram em conta os perigos. Um exemplo simplificado de uma entrada do log de perigo para o sistema de fornecimento de insulina é mostrado no Quadro 24.1. Esse formulário documenta o processo de análise de perigos e mostra os requisitos de projeto gerados durante esse processo. Esses requisitos de projeto destinam-se a garantir que o sistema de controle nunca possa fornecer uma dose excessiva a um usuário da bomba de insulina.

Conforme mostrado no Quadro 24.1, as pessoas com responsabilidades de segurança devem ser identificadas explicitamente. Os projetos de desenvolvimento de sistemas críticos de segurança devem sempre nomear um engenheiro de segurança de projeto que não esteja envolvido no desenvolvimento do sistema. A responsabilidade do engenheiro é assegurar que verificações apropriadas de segurança sejam realizadas e documentadas. O sistema pode também exigir um avaliador de segurança independente, a ser nomeado de fora da organização, para relatar assuntos de segurança diretamente ao cliente.

Em alguns domínios, os engenheiros de sistemas com responsabilidades de segurança devem ser certificados. No Reino Unido, isso significa que eles devem ter sido aceitos como membros de institutos de engenharia (civil, elétrica, mecânica etc.) e devem ser engenheiros especialistas. Engenheiros inexperientes e com pouca qualificação não podem ter a responsabilidade pela segurança.

Isso não se aplica atualmente aos engenheiros de software, embora tenha havido uma extensa discussão sobre o licenciamento de engenheiros de software em vários estados nos Estados Unidos (Knight e Leveson, 2002; Bagert, 2002). Contudo, os futuros padrões de processo para desenvolvimento de software crítico de segurança podem exigir que os engenheiros de segurança de projeto devam ser engenheiros certificados com um nível mínimo de treinamento definido.

24.2.3 Verificação de segurança em run-time

Descrevi a programação defensiva no Capítulo 20, na qual são adicionadas declarações redundantes a um programa para monitorar sua operação e verificar possíveis defeitos do sistema. A mesma técnica pode ser usada para a monitoração dinâmica de sistemas críticos de segurança. O código de verificação pode ser adicionado ao sistema que verifica uma restrição de segurança. Ele lança uma exceção caso essa restrição seja violada. As restrições de segurança, que devem

Quadro 24.1

Página de registro de perigo simplificada

Registro de Perigo **Página 4: Impressa em 20.02.2003**

Sistema: Sistema de Bomba de Insulina *Arquivo:* InsulinPump/Safety/HazardLog
Engenheiro de Segurança: James Brown *Versão do registro:* 1/3

Perigo Identificado	Dose excessiva de insulina fornecida ao paciente				
Identificado por	Jane Williams				
Classe de criticidade	1				
Risco identificado	Alto				
Árvore de defeitos identificada	SIM	*Data*	24.01.99	*Localização*	Hazard Log, Página 5
Criadores da árvore de defeitos	Jane Williams e Bill Smith				
Árvore de defeitos verificada	SIM	Data	28.01.99	*Verificador*	*James Brown*

Requisitos de projeto de segurança de sistema

1. O sistema deve incluir software de autoteste que irá testar o sistema de sensores, o relógio e o sistema de fornecimento de insulina.
2. O software de autoverificação será executado uma vez por minuto.
3. No evento de o software de autoverificação descobrir um defeito em qualquer um dos componentes do sistema, um aviso sonoro será emitido e o display da bomba deverá indicar o nome do componente no qual o defeito foi descoberto. O fornecimento de insulina deve ser suspenso.
4. O sistema incorporará um sistema de sobreposição que permite ao usuário modificar a dose de insulina calculada a ser fornecida pelo sistema.
5. A quantidade de sobreposição deve estar limitada a não ser maior do que um valor predefinido na configuração do sistema pela equipe médica.

sempre ser mantidas em determinados pontos no programa, podem ser expressas como *asserções*. As asserções são predicados que descrevem condições a serem mantidas antes que a declaração seguinte seja executada. Em sistemas críticos de segurança, as asserções devem ser geradas baseadas na especificação de segurança. Elas concentram-se mais em assegurar o comportamento seguro do que o comportamento que atenda à especificação.

As asserções podem ser particularmente valiosas para garantir a segurança das comunicações entre os componentes do sistema. Por exemplo, no sistema de fornecimento de insulina, a dose administrada envolve a geração de sinais para a bomba de insulina que fornece um número especificado de incrementos de insulina (Figura 24.8). O número de incrementos de insulina associado à máxima dose permitida pode ser pré-calculado e incluído como uma asserção no sistema.

Se houver um erro no cálculo de **currentDose**, que é a variável de estado que define a quantidade de insulina a ser fornecida, ou se esse valor for corrompido de alguma maneira, isso será detectado nesse estágio. Uma dose excessiva de insulina não será fornecida, pois o método assegura que a bomba não fornecerá mais do que **maxDose**.

Com base nessas asserções de segurança incluídas como comentários de programa, pode ser gerado o código para verificar essas asserções. Você pode ver isso na Figura 24.8, na qual a declaração if, depois do comentário afirmativo, verifica a asserção. Em princípio, grande parte da geração de código poderia ser automatizada por meio de um pré-processador de asserções. Contudo, essas ferramentas devem ser especialmente escritas e o código de asserção é normalmente gerado a mão.

Figura 24.8

Administração de insulina com verificação em tempo de execução.

```
static void administerInsulin ( ) throws SafetyException {

        int maxIncrements = InsulinPump.maxDose / 8 ;
        int increments = InsulinPump.currentDose / 8 ;

        // assert currentDose <= InsulinPump.maxDose

        if (InsulinPump.currentDose > InsulinPump.maxDose)
            throw new SafetyException (Pump.doseHigh);
        else
            for (int i=1; i<= increments; i++)
            {
                generateSignal () ;
                if (i > maxIncrements)
                    throw new SafetyException ( Pump.incorrectIncrements);
                } // for loop

    } //administerInsulin
```

24.3 Avaliação de proteção

A avaliação de proteção do sistema está se tornando cada vez mais importante à medida que cada vez mais sistemas críticos são habilitados para a Internet e podem ser acessados por qualquer pessoa conectada a uma rede. Existem relatos diários sobre ataques em sistemas baseados na Web, vírus e worms regularmente distribuídos por meio dos protocolos da Internet.

Tudo isso significa que os processos de verificação e validação para sistemas baseados na Web devem enfocar a avaliação de proteção, na qual a capacidade do sistema em resistir a diferentes tipos de ataque é testada. Contudo, conforme explica Anderson (Anderson, 2001), esse tipo de avaliação de proteção é muito difícil de realizar. Conseqüentemente, os sistemas são muitas vezes implantados com brechas de proteção pelas quais os invasores ganham acesso ou danificam esses sistemas.

Fundamentalmente, a razão por que a proteção é tão difícil de avaliar é que os requisitos de proteção, assim como alguns requisitos de segurança, são requisitos do tipo 'não deve'. Ou seja, eles especificam o que não deve ocorrer em vez da funcionalidade de sistema ou do comportamento necessário. Geralmente não é possível definir o comportamento não desejado como uma simples restrição que pode ser verificada pelo sistema.

Se os recursos estão disponíveis, você pode sempre demonstrar que um sistema atende a seus requisitos funcionais. Entretanto, é impossível provar que um sistema não faz alguma coisa; assim, independentemente da quantidade de testes, vulnerabilidades de proteção podem permanecer em um sistema depois que ele foi implantado. Mesmo em sistemas que estiveram em uso por muitos anos, um invasor engenhoso pode descobrir uma nova maneira de ataque e penetrar no que foi considerado, até então, um sistema seguro. Por exemplo, o algoritmo RSA para criptografia de dados, que foi considerado seguro por muitos anos, foi quebrado em 1999.

Existem quatro abordagens complementares para a verificação de proteção:

1. *Validação baseada em experiências*. Neste caso, o sistema é analisado em relação aos tipos de ataque conhecidos pela equipe de validação. Esse tipo de validação é geralmente realizado em conjunto com a validação baseada em ferramentas. Checklists de problemas de proteção conhecidos (Quadro 24.2) podem ser criadas para ajudar no processo. Essa abordagem pode usar toda a documentação do sistema e ser parte de outras revisões de sistema que verificam erros e omissões.
2. *Validação baseada em ferramentas*. Nesse caso, várias ferramentas de proteção, como verificadores de senhas, são usados para analisar o sistema. Verificadores de senhas detectam senhas inseguras, como nomes comuns ou strings de letras consecutivas. Isso é, realmente, uma extensão da validação baseada em experiências na qual a experiência é incorporada às ferramentas usadas.
3. *Equipes 'tigres'*. Nesse caso, uma equipe é constituída com o objetivo de violar a proteção do sistema. Eles simulam ataques ao sistema e usam sua engenhosidade para descobrir novas maneiras de comprometer a proteção do sistema. Essa abordagem pode ser muito eficaz, especialmente se os membros da equipe têm experiência em invadir sistemas.
4. *Verificação formal*. Um sistema pode ser verificado em relação a uma especificação formal de proteção. Contudo, como em outras áreas, a verificação formal de proteção não é amplamente usada.

É muito difícil para os usuários finais verificar a proteção de um sistema. Conseqüentemente, conforme explicado por Gollmann (Gollmann, 1999), entidades na América do Norte e na Europa estabeleceram conjuntos de critérios de avaliação

Quadro 24.2

Exemplos de entradas em checklist de proteção.

Checklist de proteção

1. Todos os arquivos que são criados na aplicação têm permissões de acesso apropriadas? As permissões de acesso errado podem levar esses arquivos a serem acessados por usuários não autorizados.
2. O sistema termina automaticamente as sessões de usuário depois de um período de inatividade? Sessões que são deixadas ativas podem permitir acessos não autorizados por computadores não monitorados.
3. Se o sistema é escrito em uma linguagem de programação sem verificação de limites de vetores, existem situações em que o overflow de buffer pode ser explorado? Overflow de buffer pode permitir aos agressores enviar strings de códigos ao sistema e então executá-los.
4. Se as senhas estão definidas, o sistema verifica se as senhas são fortes? Senhas fortes consistem em letras, números e pontuações misturadas, e não são entradas normais de dicionários. Eles são mais difíceis de quebrar do que senhas simples.

de proteção que podem ser verificados por avaliadores especializados. Fornecedores de produtos de software podem submeter seus produtos para avaliação e certificação em relação a esses critérios.

Portanto, se você tiver um requisito para determinado nível de proteção, pode escolher um produto validado para esse nível. Contudo, muitos produtos não são certificados quanto à proteção ou sua certificação aplica-se somente a produtos individuais. Quando o sistema certificado é usado em conjunto com outros sistemas não certificados, como software desenvolvido localmente, o nível de proteção total do sistema não pode ser avaliado.

24.4 Casos de segurança e confiabilidade

Casos de segurança e, mais genericamente, casos de confiabilidade são documentos estruturados que estabelecem argumentos detalhados e evidências de que um sistema é seguro ou de que um nível de confiabilidade exigido foi atingido. Para muitos tipos de sistemas críticos, a produção de um caso de segurança é um requisito legal e o caso deve satisfazer alguma certificação antes que o sistema seja implantado.

Órgãos regulamentadores são criados pelos governos para assegurar que a iniciativa privada não se beneficie deixando de seguir padrões nacionais de segurança, proteção etc. Existem órgãos regulamentadores em muitos setores diferentes. Por exemplo, as empresas aéreas são regulamentadass pelas autoridades nacionais de aviação, como a FAA (nos EUA) e a CAA (no Reino Unido). Existem órgãos regulamentadores para assegurar a segurança das ferrovias e órgãos regulamentadores da indústria nuclear devem se certificar da segurança de uma instalação nuclear antes que ela entre em operação. Na área financeira, os bancos nacionais funcionam como órgãos regulamentadores, definindo procedimentos e práticas para reduzir a possibilidade de fraudes e para proteger os clientes dos bancos de práticas arriscadas. À medida que os sistemas de software se tornaram cada vez mais importantes nas infra-estruturas críticas dos países, esses órgãos regulamentadores se tornaram cada vez mais dedicados aos casos de segurança e confiabilidade de sistemas de software.

O papel de um órgão regulamentador é verificar se um sistema finalizado é tão seguro quanto prático e, assim, esse envolvimento se dá principalmente quando um projeto de desenvolvimento está completo. Contudo, os regulamentadores e os desenvolvedores raramente trabalham isolados; eles se comunicam com a equipe de desenvolvimento para estabelecer o que deve ser incluído no caso de segurança. O regulamentador e os desenvolvedores examinam conjuntamente processos e procedimentos para assegurar que eles estão aprovados e documentados de acordo com o órgão regulamentador.

Naturalmente, o software em si não é perigoso. Ele é perigoso somente quando está incorporado a um sistema de grande porte, baseado em computador ou sociotécnico, no qual as falhas de software podem resultar em falhas de outros equipamentos ou processos que podem causar prejuízos ou morte. Portanto, um caso de segurança de software é sempre parte de um caso de segurança de sistema mais amplo que demonstra a segurança do sistema total. Ao criar um caso de segurança de software, você deve relacionar as falhas de software a falhas maiores de sistema e demonstrar que essas falhas de software não ocorrerão ou que elas não serão propagadas de maneira que falhas perigosas de sistema possam ocorrer.

Um caso de segurança é um conjunto de documentos que incluem uma descrição do sistema que deve ser certificado, informações sobre o processo usado para desenvolver o sistema e, criticamente, argumentos lógicos que demonstrem que o sistema provavelmente é seguro. Mais sucintamente, Bishop e Bloomfield (Bishop e Bloomfield, 1998; Bishop e Bloomfield, 1995) definem um caso de segurança como:

> *Um conjunto de evidências documentadas que fornece um argumento convincente e válido de que o sistema é adequadamente seguro para determinada aplicação em determinado ambiente.*

A organização e o conteúdo de um caso de segurança dependem do tipo de sistema que será certificado e de seu contexto de operação. A Tabela 24.1 mostra uma possível organização para um caso de segurança de software.

Um componente principal de um caso de segurança é um conjunto de argumentos lógicos para a segurança do sistema. Estes podem ser argumentos absolutos (determinado evento ocorrerá ou não) ou argumentos probabilísticos (a probabilidade de determinado evento ocorrer é 0,Y); quando combinados, eles devem demonstrar segurança. Conforme mostrado na Figura 24.9, um argumento é um relacionamento entre a idéia do caso (uma declaração) e um conjunto de evidências coletadas. O argumento explica essencialmente por que a declaração (que é geralmente de se o sistema é seguro) pode ser inferida baseada em evidência disponível. Naturalmente, dada a natureza de múltiplos níveis de um sistema, as declarações são organizadas em uma hierarquia. Para demonstrar que uma declaração de alto nível é válida, você deve primeiro refinar os argumentos para declarações de níveis mais baixos. A Figura 24.10 mostra parte dessa hierarquia de declarações para o sistema de bomba de insulina.

Como um dispositivo médico, o sistema de bomba de insulina deve ser certificado externamente. Por exemplo, no Reino Unido, a entidade regulamentadora, Medical Devices Directorate, deve emitir um certificado de segurança para qualquer dispositivo médico vendido no Reino Unido. Vários argumentos devem ser apresentados para demonstrar a segurança desse sistema. Por exemplo, o seguinte argumento poderia fazer parte de um caso de segurança para a bomba de insulina.

Tabela 24.1 Componentes de um caso de segurança de software

Componente	Descrição
Descrição do sistema	Uma visão geral do sistema e uma descrição de seus componentes críticos
Requisitos de segurança	Os requisitos de segurança abstraídos da especificação de requisitos do sistema
Análise de perigos e de riscos	Documentos que descrevem perigos e riscos identificados e as medidas tomadas para reduzir os riscos
Análise de projeto	Um conjunto de argumentos estruturados que justificam por que o projeto é seguro
Verificação e validação	Uma descrição de procedimentos de V & V usados e, quando apropriado, os planos de teste para o sistema
Relatórios de revisão	Registros de todas as revisões de projeto e de segurança
Competência de equipe	Evidência de competência de toda a equipe envolvida no desenvolvimento e validação de sistemas relacionados à segurança
Processo de GQ	Registros de processos de garantia de qualidade produzidos durante o desenvolvimento de sistema
Processos de gerenciamento de mudanças	Registros de todas as mudanças propostas, ações tomadas e, quando apropriado, justificativas de segurança dessas mudanças
Casos de segurança associados	Referências a outros casos de segurança que podem influenciar este caso de segurança

Figura 24.9
Estrutura de um argumento.

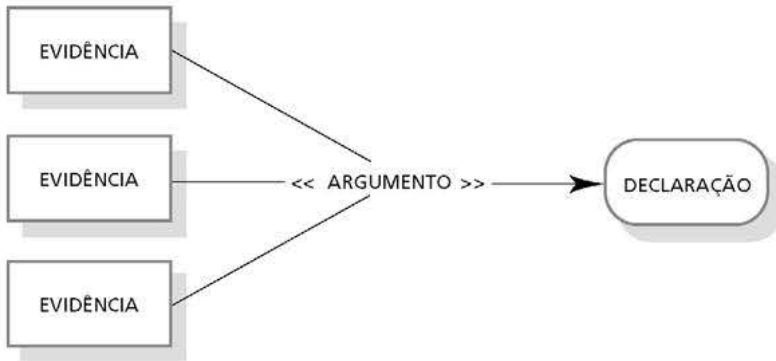
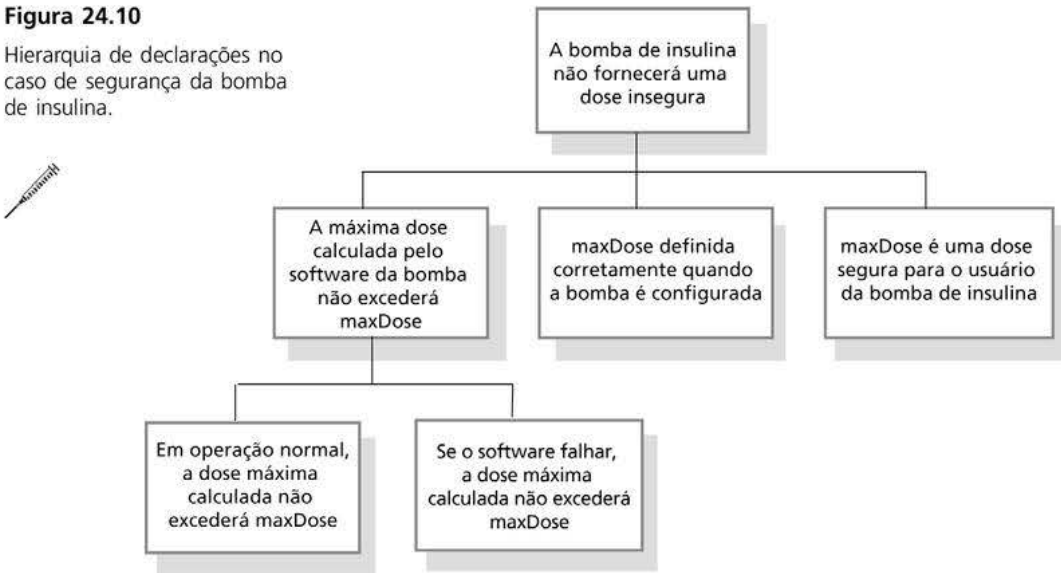


Figura 24.10
Hierarquia de declarações no caso de segurança da bomba de insulina.



Declaração: A dose máxima calculada pela bomba de insulina não excederá **maxDose**.

Evidência: Argumento de segurança para bomba de insulina (Figura 24.7).

Evidência: Conjuntos de dados de teste para a bomba de insulina.

Evidência: Relatório de análise estática para o software da bomba de insulina.

Argumento: O argumento de segurança apresentado mostra que a dose máxima de insulina que pode ser calculada é igual a **maxDose**.

Em 400 testes o valor de **Dose** foi corretamente calculado e nunca excedeu **maxDose**.

A análise estática do software de controle não revelou anomalias.

Em geral, é razoável pressupor que a declaração é justificada.

Naturalmente, este é um argumento muito simplificado e, em um caso de segurança real, seriam apresentadas referências detalhadas para as evidências. Devido a sua natureza detalhada, os casos de segurança são documentos muito longos e complexos. Várias ferramentas de software estão disponíveis para ajudar em sua construção e incluem links para essas ferramentas no site Web do livro.

PONTOS-CHAVE

- O teste estatístico é usado para estimar a confiabilidade do software. Ele consiste em testar o sistema com um conjunto de dados de teste que reflete o perfil operacional do software. Dados de teste podem ser gerados automaticamente.
- Modelos de aumento da confiabilidade apresentam a mudança na confiabilidade quando os defeitos são removidos do software durante o processo de teste. Os modelos de confiabilidade podem ser usados para prever quando a confiabilidade exigida será atingida.
- As provas de segurança são uma técnica eficaz de garantia de segurança do produto. Elas mostram que uma condição perigosa identificada nunca pode ocorrer. São geralmente mais simples do que provar que um programa atende a sua especificação.
- É importante ter um processo bem definido e certificado para o desenvolvimento de sistemas críticos de segurança. O processo deve incluir a identificação e a monitoração de perigos potenciais.
- A validação de segurança pode ser realizada por meio da análise baseada em experiências, análise baseada em ferramentas ou em equipes 'tigres' que simulam ataques ao sistema.
- Os casos de segurança coletam todas as evidências que demonstram que um sistema é seguro. Esses casos de segurança são exigidos quando um regulamentador externo deve certificar o sistema antes que ele seja usado.

LEITURAS SUGERIDAS

"Best practices in code inspection for safety-critical software". Este artigo prático apresenta um checklist de diretrizes para inspeção e revisão de software crítico de segurança (J. R. de Almeida, et al., *IEEE software*, 20(3), maio/junho de 2003).

"Statically scanning Java code: finding security vulnerabilities". Este é um bom artigo sobre como evitar vulnerabilidades de segurança em geral. Ele explica como essas vulnerabilidades surgem e como podem ser detectadas usando um analisador estático (J. Viega, et al., *IEEE software*, 17(5), setembro/outubro de 2000).

Software reliability engineering: more reliable software, faster development and testing. Este é, provavelmente, o livro definitivo sobre o uso de perfis operacionais e modelos para avaliação de confiabilidade. Ele inclui detalhes de experiências com testes estatísticos (J. D. Musa, 1998, McGraw-Hill.)

Safety-critical computer systems. Este excelente livro inclui um capítulo particularmente bom sobre a posição de métodos formais no desenvolvimento de sistemas críticos de segurança (N. Storey, 1996, Addison-Wesley.)

Safeware: system safety and computers. Este livro inclui um bom capítulo sobre a validação de sistemas críticos de segurança com mais detalhes do que apresentei aqui sobre o uso de argumentos de segurança baseados em árvores de defeitos (N. Leveson, 1995, Addison-Wesley.)

EXERCÍCIOS

24.1 Descreva como você faria a especificação de validação de confiabilidade do sistema de supermercado que você especificou no Exercício 9.8. Sua resposta deve incluir uma descrição de quaisquer ferramentas que possam ser usadas.

24.2 Explique por que é praticamente impossível validar especificações de confiabilidade quando estas são expressas em termos de um número muito pequeno de falhas durante o tempo de vida total de um sistema.

24.3 Usando a literatura como informações de apoio, escreva um relatório para a gerência (que não tem nenhuma experiência prévia nessa área) sobre o uso de modelos de crescimento de confiabilidade.

24.4 É ético para um engenheiro concordar em entregar um sistema de software com defeitos conhecidos para um cliente? Faz alguma diferença se o cliente for comunicado antecipadamente sobre a existência desses defeitos? Seria razoável fazer declarações sobre a confiabilidade do software nessas circunstâncias?

24.5 Explique por que garantir a confiabilidade do sistema não é uma garantia de segurança do sistema.

24.6 O mecanismo de controle de fechadura da porta em uma área de depósito de lixo nuclear é projetado para fornecer uma operação segura. Ele garante que a entrada para o depósito seja somente permitida quando os anteparos de radiação estiverem posicionados ou quando o nível de radiação no depósito cair abaixo de determinado valor (**dangerLevel**). Isto é:

a. Se os anteparos de radiação controlados remotamente estiverem posicionados dentro de uma sala, a porta poderá ser aberta por um operador autorizado.

b. Se o nível de radiação na sala estiver abaixo de um valor especificado, a porta poderá ser aberta por um operador autorizado.

c. Um operador autorizado é identificado por meio de um código de entrada autorizado.

O código Java mostrado na Figura 24.11 controla o mecanismo de fechamento da porta. Observe que o estado seguro é aquele no qual a entrada não deve ser permitida. Desenvolva um argumento de segurança que mostre que este código é potencialmente inseguro. Modifique o código para torná-lo seguro.

24.7 Usando a especificação para o cálculo de dosagem no Capítulo 10 (Figura 10.10), escreva o método **computeInsulin** em Java, conforme usado no Figura 24.6. Crie um argumento informal de segurança de que esse código é seguro.

24.8 Sugira como você faria a validação de um sistema de proteção de senhas para uma aplicação que você tenha desenvolvido. Explique a função de quaisquer ferramentas que você imagina que possam ser úteis.

24.9 Por que é necessário incluir detalhes de mudanças de sistema em um caso de segurança de software?

24.10 Liste quatro tipo de sistemas que exigiriam casos de segurança de software de sistema.

24.11 Suponha que você fez parte da equipe que desenvolveu um software para uma indústria química, o qual falhou de alguma maneira causando um sério incidente de poluição. Seu chefe é entrevistado na televisão e declara que o processo de validação é abrangente e que não existem defeitos no software. Ele afirma que o problema deve ser devido aos procedimentos operacionais inadequados. Um jornal quer saber a sua opinião. Explique como você deve lidar com tal entrevista.

24.12 Quais são os argumentos a favor e contra a licença formal de engenheiros de software?

Figura 24.11

Controlador de fechadura de porta.

```
entryCode = lock.getEntryCode () ;
if (entryCode == lock.authorisedCode)
{
    shieldStatus = Shield.getStatus ();
    radiationLevel = RadSensor.get ();
    if (radiationLevel < dangerLevel)
        state = safe;
    else
        state = unsafe;
    if (shieldStatus == Shield.inPlace() )
        state = safe;
    if (state == safe)
        {
            Door.locked = false ;
            Door.unlock ();
        }
    else
    {
        Door.lock ( );
        Door.locked := true ;
    }
}
```

PARTE 6 GERENCIAMENTO

Algumas vezes, sugere-se que a diferença principal entre engenheiros de software e programadores é que a engenharia de software é um processo gerenciado. Por isso, entendo que o desenvolvimento de software ocorre dentro de uma organização e está sujeito a uma série de cronogramas e a restrições organizacionais e de orçamento. Você pode diferenciar essa situação com o desenvolvimento open-source, que é fundamentalmente uma atividade voluntária. Os desenvolvedores open-source tomam suas decisões sobre quando e como trabalham no software, e a qualidade de seu trabalho mostra que o desenvolvimento de software bem-sucedido não deve ser rigorosamente gerenciado.

Entretanto, a maioria dos projetos de software não pode ser desenvolvida usando uma abordagem open-source — esse modelo de desenvolvimento é adequado somente a sistemas de infra-estrutura, como sistemas operacionais, servidores Web, compiladores etc. O desenvolvimento de aplicações especializadas é sempre um processo gerenciado. Os capítulos desta parte do livro estendem o capítulo introdutório sobre gerenciamento na Visão Geral (Capítulo 5) e explicam uma série de tópicos de gerenciamento:

1. O Capítulo 25 dedica-se ao gerenciamento de pessoal. Esse não é um tópico técnico, de modo que normalmente não é abordado em livros de engenharia de software. Contudo, em minha experiência, o gerenciamento de pessoal é uma atividade crítica de gerenciamento em projetos de desenvolvimento de software. Meu objetivo aqui é introduzir as questões e os problemas de gerenciamento de pessoal. Abordo a seleção e motivação de pessoal, gerenciamento de grupos de projeto e, finalmente, o Modelo de Maturidade de Capacitação de Pessoal da SEI.

2. No Capítulo 26 enfoco a estimativa de custo de software. Explico a produtividade de software e várias técnicas de estimativa de custos. Existe uma grande incerteza nessa área e algumas pessoas acreditam que o melhor meio de enfrentar isso é por meio de modelagem algorítmica de custos. Abordo o modelo COCOMO II nesse capítulo, mas, como é um modelo muito abrangente, posso somente fornecer uma breve introdução de suas características fundamentais.

3. Os capítulos 27 e 28 são dedicados às questões sobre gerenciamento de qualidade. O gerenciamento de qualidade concentra-se nos processos e técnicas para assegurar e aprimorar a qualidade do software, e o Capítulo 27 introduz o assunto. O Capítulo 28 retorna a esse assunto com uma discussão sobre o aprimoramento do processo de software — como os processos podem ser alterados para que tanto os atributos de processo como os de produto sejam aprimorados.

4. Finalmente, o Capítulo 29 explica o gerenciamento de configuração. Essa é uma questão importante para todos os sistemas de grande porte desenvolvidos por equipes. Contudo, a necessidade de gerenciamento de configuração nem sempre é óbvia para estudantes que se dedicaram ao desenvolvimento de software pessoal, de modo que eu apresento os vários aspectos desse tópico nesse capítulo, entre eles planejamento de configuração, gerenciamento de versões, gerenciamento de construção e de mudança de sistema.

25

Gerenciamento de pessoal

Objetivos

O objetivo deste capítulo é explicar a importância das pessoas no processo de engenharia de software. Após ler este capítulo, você:

- compreenderá algumas das questões envolvidas na seleção e retenção de pessoal em uma organização de desenvolvimento de software;
- compreenderá fatores que influenciam a motivação de pessoas e suas implicações para os gerentes de projeto de software;
- compreenderá as questões principais de trabalho em equipe, entre elas composição, coesão, comunicações e organização de equipes;
- estará ciente da estrutura do Modelo de Maturidade de Capacitação de Pessoal — modelo que é um framework para aprimorar as capacidades dos engenheiros de software em uma organização.

Conteúdo

25.1 Seleção de pessoal
25.2 Motivação de pessoas
25.3 Gerenciamento de grupos
25.4 Modelo de Maturidade de Capacitação de Pessoal

As pessoas são o maior patrimônio de uma organização de software. Elas representam o capital intelectual, e é responsabilidade dos gerentes de software garantir que a organização obtenha o melhor retorno sobre o investimento em pessoas. Em empresas e economias bem-sucedidas, isso é alcançado quando as pessoas são respeitadas pela organização. Elas devem ter um nível de responsabilidade e remuneração que reflitam suas habilidades.

O gerenciamento eficiente está, portanto, relacionado ao gerenciamento de pessoas em uma organização. Os gerentes de projeto devem resolver problemas técnicos e não técnicos usando as pessoas de sua equipe da maneira mais eficiente possível. Devem motivar as pessoas, planejar e organizar o trabalho para assegurar que este seja feito apropriadamente. O gerenciamento inadequado de pessoas é uma das mais significativas contribuições para a falha do projeto.

Infelizmente, a fraca liderança é muito comum no setor de software. Os gerentes falham em levar em conta as limitações das pessoas e impõem prazos irreais às equipes de projeto. Eles gerenciam por meio de reuniões, porém falham em permitir que as pessoas contribuam com o projeto nessas reuniões. Eles aceitam novos requisitos sem análise apropriada do que significam para a equipe de projeto. Algumas vezes, eles se vêem como exploradores do pessoal em vez de trabalharem com eles para identificar como seu trabalho pode contribuir para as metas organizacionais e pessoais.

Existem, na minha opinião, quatro fatores críticos no gerenciamento de pessoal:

1. *Consistência*. A pessoas de uma equipe de projeto devem ser tratadas de maneira uniforme. Mesmo que ninguém espere que todas as remunerações sejam idênticas, as pessoas não devem sentir que sua contribuição para a organização é desvalorizada.
2. *Respeito*. Pessoas diferentes têm habilidades diferentes e os gerentes devem respeitar essas diferenças. Deve ser dada uma oportunidade a todos os membros da equipe para fazer uma contribuição. Em alguns casos, naturalmente, você encontrará pessoas que simplesmente não se ajustam a uma equipe e não podem continuar, mas é importante não ignorar as conclusões sobre isso.
3. *Inclusão*. As pessoas contribuem efetivamente quando sentem que os outros ouvem e levam em conta suas propostas. É importante desenvolver um ambiente de trabalho no qual todos os pontos de vista, mesmo aqueles do pessoal mais jovem, sejam considerados.
4. *Honestidade*. Como gerente, você deve ser sempre honesto sobre o que vai ou não bem na equipe. Você deve também ser honesto sobre seu nível de conhecimento técnico e estar disposto a dar mais informações à equipe quando necessário. Se não for honesto, isso será eventualmente descoberto e você perderá o respeito do grupo.

Meu objetivo neste capítulo é introduzir algumas das questões que os gerentes de engenharia de software podem encontrar e fornecer informações básicas que ajudem a compreender essas questões. O gerenciamento, em meu ponto de vista, é alguma coisa que somente pode ser aprendida por meio da experiência, de modo que a função de um livro é ajudá-lo a aprender a partir de experiências anteriores. Você não se tornará um bom gerente de pessoal simplesmente lendo este capítulo. Entretanto, tenho a esperança de que este material ajude a compreender os problemas que os gerentes enfrentam ao lidarem com equipes compostas de pessoas tecnicamente talentosas.

25.1 Seleção de pessoal

Uma das mais importantes tarefas de gerenciamento de projeto é a de seleção de equipe. Em casos excepcionais, os gerentes de projeto podem nomear pessoas que sejam as mais adequadas para o trabalho independentemente de suas outras responsabilidades ou considerações de orçamento. Muito freqüentemente, contudo, os gerentes de projeto não têm livre escolha de pessoal. Eles podem precisar usar quem esteja disponível na organização, podem precisar encontrar pessoas muito rapidamente e podem ter um orçamento limitado. As limitações de orçamento podem restringir o número de engenheiros experientes, dispendiosos e disponíveis para trabalhar no projeto.

A decisão sobre quem nomear para o projeto é tomada geralmente usando três tipos de informações:

1. Informações fornecidas pelos candidatos sobre sua formação e experiência (seu currículo). Esta é, geralmente, a evidência mais confiável disponível para julgar se os candidatos serão úteis.
2. Informações obtidas pelas entrevistas de candidatos. As entrevistas podem dar a você uma boa impressão a respeito de se um candidato é um bom comunicador e se ele ou ela tem boas habilidades sociais. Contudo, algumas pesquisas mostraram que os entrevistadores estão sujeitos a aceitar ou rejeitar candidatos com base em julgamentos rápidos e subjetivos. Conseqüentemente, as entrevistas não constituem métodos confiáveis para fazer julgamentos sobre capacidades técnicas.
3. Recomendações de pessoas que trabalharam com os candidatos. Isso pode ser eficaz quando você conhece as pessoas que fazem a recomendação. Por outro lado, as recomendações podem não ser confiáveis e, em meu ponto de vista, são pouco úteis na tomada de decisão sobre o pessoal.

O Quadro 25.1 apresenta um pequeno estudo de caso sobre questões que surgem quando da nomeação de pessoal. Algumas lições características desse estudo de caso são:

1. Se você está procurando pessoas com habilidades específicas dentro da própria empresa, o gerente do projeto dessas pessoas pode não desejar perdê-las. Algumas vezes, você precisa aceitar pessoas que serão capazes de trabalhar somente parte do tempo em seu projeto.
2. Habilidades como o projeto de interface com o usuário e interfaces de hardware têm baixa oferta. Você pode não ter muitas opções de candidatos para esses cargos, especialmente se sua empresa não ficar próxima de outras empresas de software.
3. Os recém-formados podem não ter as habilidades específicas de que você necessita, mas eles normalmente estão entusiasmados e podem conhecer as tecnologias mais recentes.

Quadro 25.1

Seleção de pessoal.

Estudo de caso 1: A escolha dos membros da equipe

Alice é uma gerente de projeto de software funcionária de uma empresa que desenvolve sistemas de alarme. Essa empresa deseja entrar no crescente mercado de tecnologia de assistência a pessoas idosas ou incapacitadas de viver independentemente. Alice foi indicada para liderar uma equipe de seis desenvolvedores que podem trabalhar em novos produtos baseados na tecnologia de alarme da empresa. Sua primeira tarefa é selecionar os membros da equipe entre engenheiros de software que faziam parte da empresa e também de fora dela.

Para ajudar na seleção da equipe, primeiro Alice deve avaliar as habilidades de que necessita. São elas:

- Experiência com a tecnologia de alarme existente, uma vez que será reusada.
- Experiência em projetos de interface de usuário, pois os usuários não são treinados e podem estar incapacitados; sendo assim, necessitam de recursos como tamanhos variáveis de fontes de letras etc.
- De maneira ideal, uma pessoa com experiência em projetos de sistemas de tecnologia de assistência. Por outro lado, uma pessoa com experiência em interfaces de unidades de hardware, uma vez que todos os sistemas a serem desenvolvidos envolvem algum controle de hardware.
- Habilidades gerais de desenvolvimento.

O próximo estágio é tentar e encontrar pessoas de dentro da empresa com as habilidades necessárias, embora a empresa tenha se expandido significativamente e tenha pouco pessoal disponível. O melhor que Alice pode obter é a ajuda de um especialista em alarmes (Fred) por 2 dias da semana. Ela então decide publicar um anúncio para encontrar o novo pessoal do projeto, listando os atributos de que ela precisa:

- Experiência em programação em C. Ela decide desenvolver todo o software de controle da tecnologia de assistência em C.
- Experiência em projetos de interface com o usuário. Um projetista de interface com o usuário é essencial, mas pode não ser necessária uma alocação em tempo integral.
- Experiência em interfaces de hardware com C e uso de sistemas de desenvolvimento remotos. Todos os dispositivos usados têm interfaces complexas de hardware.
- Experiência de trabalho com engenheiros de hardware. Às vezes, será necessário construir um hardware completamente novo.
- Personalidade solidária para se relacionar e trabalhar com pessoas idosas que estão fornecendo requisitos e testando o sistema.

Alice obtém 30 respostas para o anúncio e, dentre os candidatos, é capaz de identificar candidatos adequados para interface de hardware (Dorothy) e com experiência em projetos de interface (Ed). Ela também decide contratar dois recém-formados (Brian e Bob) com alguma experiência em programação em C, mas que devem essencialmente ser treinados na empresa. Quanto ao restante, a solução é nomear um programador mais experiente para se juntar à equipe de desenvolvimento. Alice tem duas opções. Carol tem vários anos de experiência em programação em C e, recentemente, interrompeu a carreira para ter um filho. Dave tem experiência em programação comparável à de Carol e é um entusiasta em programação. Ele gasta seu tempo livre trabalhando em projetos de desenvolvimento open-source e tem vasto conhecimento de C e C++.

Depois de entrevistar Carol e Dave, Alice decide oferecer o trabalho para Carol, embora Dave tenha um profundo conhecimento de programação.

4. Nem sempre há sentido em empregar a pessoa mais proficiente tecnicamente em um trabalho de desenvolvimento técnico. Pode ser que a interação com usuários idosos seja exigida e Alice acha que Carol é mais solidária aos problemas dessas pessoas.

Se você tiver alguma escolha de pessoal a fazer, os fatores que podem influenciar em sua decisão são mostrados na Tabela 25.1. Os fatores mais importantes variam dependendo do domínio da aplicação, do tipo de projeto e das habilidades dos outros membros da equipe de projeto.

Como um gerente de projetos, você pode estar diante de problemas para encontrar pessoas com habilidades adequadas e experiência. Você pode precisar montar sua equipe usando engenheiros relativamente inexperientes. Isso pode trazer problemas, pois os membros da equipe não compreendem o domínio da aplicação ou a tecnologia. Entretanto, em um projeto de longo prazo, a compreensão de conceitos e de domínio da aplicação é mais importante do que conhecimento específico, particularmente de linguagens de programação e métodos. Portanto, a menos que você necessite de um conhecimento específico de uma linguagem de programação para uma atribuição de curto prazo, é melhor selecionar o pessoal com experiência em resolver problemas ou experiência no domínio. É relativamente fácil aprender uma nova linguagem, mas é muito mais difícil desenvolver o conhecimento conceitual mais profundo exigido para resolver problemas complexos.

Algumas empresas testam os candidatos às equipes. Isso inclui testes de aptidão em programação e testes psicométricos nos quais os candidatos completam uma série de pequenos exercícios em um período de tempo relativamente curto. Testes

Tabela 25.1 Fatores que direcionam a seleção de pessoal

Fator	Explicação
Experiência no domínio da aplicação	Para que um projeto desenvolva um sistema bem-sucedido, os desenvolvedores devem compreender o domínio da aplicação. É essencial que alguns membros da equipe tenham alguma experiência no domínio.
Experiência em plataforma	Isso pode ser importante caso programação em baixo nível esteja envolvida. Por outro lado, este não é um atributo crítico.
Experiência em linguagem de programação	Isso normalmente só é importante para projetos de curta duração, quando não há tempo suficiente para aprender uma nova linguagem. Enquanto aprender uma linguagem em si não é difícil, são necessários vários meses para se tornar proficiente no uso de bibliotecas e componentes associados.
Habilidade em resolver problemas	Isso é muito importante para os engenheiros de software que, constantemente, precisam resolver problemas técnicos. Contudo, é quase impossível julgar sem conhecer o trabalho de um membro potencial da equipe.
Formação educacional	Isso pode fornecer um indicador do que o candidato conhece e de sua capacidade de aprender. Esse fator se torna cada vez mais irrelevante à medida que os engenheiros ganham experiência ao longo de vários projetos.
Capacidade de comunicação	O pessoal do projeto deve ser capaz de se comunicar verbalmente e por escrito com outros engenheiros, gerentes e clientes.
Capacidade de adaptação	A capacidade de adaptação pode ser julgada pela observação da experiência dos candidatos. Esse é um atributo importante, pois indica a capacidade de aprender.
Atitude	O pessoal do projeto deve ter uma atitude positiva a respeito de seu trabalho e ter uma boa disposição para aprender novas habilidades. Esse é um atributo importante, mas, muitas vezes, de difícil avaliação.
Personalidade	Esse é um atributo importante, mas também de difícil avaliação. Os candidatos devem ser razoavelmente compatíveis com outros membros da equipe. Nenhum tipo específico de personalidade é mais ou menos adequado à engenharia de software.

psicométricos destinam-se a traçar um perfil psicológico do candidato indicando atitudes e sua adequação a certos tipos de tarefas. Alguns gerentes consideram esses testes inúteis, outros pensam que eles fornecem informações úteis para a seleção de pessoal. Duvido que os testes de aptidão forneçam informações úteis sobre a habilidade de resolver problemas. A resolução de problemas complexos é um processo iterativo e de longo prazo. Não acredito que as habilidades necessárias para resolver problemas complexos sejam comparáveis às habilidades necessárias para completar um simples quebra-cabeças de aptidão.

A falta de pessoas técnicas com experiência pode ser o resultado de uma política organizacional na qual o pessoal tecnicamente capacitado atinge rapidamente o topo da carreira. Para progredir mais, eles devem se tornar gerentes, e essa promoção inevitavelmente significa que capacitações técnicas úteis serão perdidas. Para evitar essa perda, algumas grandes empresas desenvolveram estruturas paralelas de carreiras, técnica e gerencial, com igual valorização. As pessoas tecnicamente experientes são remuneradas no mesmo nível que os gerentes. À medida que a carreira individual se desenvolve, o profissional pode se especializar em atividades técnicas ou gerenciais, podendo passar de uma a outra sem perda de status ou de salário.

25.2 Motivação de pessoas

Um dos papéis do gerente de projeto é motivar as pessoas que trabalham com ele. Motivação significa organização do trabalho e ambiente de trabalho de forma que as pessoas se sintam estimuladas a trabalhar tão eficientemente quanto possível. Se as pessoas não estão motivadas, elas não terão interesse no trabalho que estão fazendo. Elas trabalharão lentamente, serão mais propensas a cometer erros e não contribuirão para as metas mais amplas da equipe ou da organização.

Maslow (Maslow, 1954) sugere que pessoas são motivadas pela satisfação de suas necessidades e que estas são organizadas em uma série de níveis, conforme mostrado na Figura 25.1. Os níveis mais baixos dessa hierarquia representam as necessidades fundamentais como se alimentar, dormir etc., bem como a necessidade de se sentirem seguras em um ambiente. As necessidades sociais se referem à necessidade de se sentir parte de um grupo social. As necessidades de auto-estima se referem à necessidade de se sentir respeitado pelos outros, e as necessidades de auto-realização se referem ao desenvolvi-

Figura 25.1

Hierarquia de necessidades humanas.

mento pessoal. As prioridades humanas são satisfazer as necessidades de nível baixo, como a fome, antes das necessidades mais abstratas, de nível mais alto.

As pessoas que trabalham em organizações de desenvolvimento de software geralmente não estão famintas ou sedentas e não se sentem fisicamente ameaçadas pelo seu ambiente. Portanto, assegurar as necessidades de satisfação social, auto-estima e auto-realização é mais importante do ponto de vista do gerenciamento.

1. Para satisfazer as necessidades sociais, você precisa dar às pessoas tempo para se reunirem com seus colegas de trabalho e fornecer lugares para que eles se encontrem. Isso é relativamente fácil quando todos os membros de uma equipe de desenvolvimento trabalham em um mesmo lugar, porém, cada vez mais, os membros da equipe não estão localizados no mesmo prédio ou sequer na mesma cidade ou estado. Eles podem trabalhar em diferentes organizações ou em casa, na maior parte do tempo. Comunicações eletrônicas, como e-mail e teleconferência, podem ser usadas para apoiar esse trabalho remoto. Entretanto, minha experiência com comunicações eletrônicas é que elas não satisfazem as necessidades sociais. Se sua equipe está dispersa, você deve organizar reuniões pessoais periódicas de modo que as pessoas possam interagir diretamente com os outros membros da equipe. Por meio dessa interação direta, as pessoas se tornam parte de um grupo social e podem se sentir motivadas pelas metas ou prioridades desse grupo.
2. Para satisfazer as necessidades de auto-estima, você precisa mostrar para as pessoas que elas são valorizadas pela organização. O reconhecimento público de realizações é uma maneira simples e eficaz de fazer isso. Obviamente, as pessoas devem também sentir que elas são remuneradas de acordo com um nível que reflita suas habilidades e experiência.
3. Finalmente, para satisfazer as necessidades de auto-realização, você precisa dar às pessoas responsabilidade por seu trabalho, atribuindo-lhes tarefas difíceis (não impossíveis) e fornecendo um programa de treinamento no qual elas possam desenvolver suas habilidades.

No Quadro 25.2 ilustro um problema de motivação que os gerentes comumente enfrentam. Nesse exemplo, um competente membro do grupo perde o interesse pelo trabalho e pelo grupo como um todo. A qualidade de seu trabalho cai e torna-se inaceitável. Essa situação deve ser tratada rapidamente. Se você não resolver esse problema, os outros membros do grupo ficarão insatisfeitos e sentirão que estão dividindo injustamente o trabalho.

Quadro 25.2

Motivação de pessoas.

Estudo de caso 2: Motivação

O projeto de tecnologia de assistência de Alice começa bem. Bons relacionamentos de trabalho dentro da equipe e idéias criativas são desenvolvidos. A empresa decide iniciar um sistema de mensagens ponto a ponto usando televisores digitais ligados a uma rede de alarmes para comunicações. Contudo, em alguns meses de projeto, Alice observa que Dorothy, a especialista em projetos de hardware, começa a chegar tarde, a qualidade de seu trabalho cai e, cada vez mais, ela tem dificuldade em se comunicar com outros membros da equipe.

Alice conversa sobre o problema informalmente com outros membros da equipe para tentar descobrir se as circunstâncias pessoais de Dorothy mudaram e se isso pode estar afetando seu trabalho. Eles não sabem de nada e, assim, Alice decide conversar com Dorothy para tentar compreender o problema.

Depois de negar que havia um problema, Dorothy admite que perdeu o interesse no trabalho. Ela esperava que fosse capaz de desenvolver e usar suas habilidades com interface de hardware. Entretanto, devido à direção escolhida para o produto, ela tem pouca oportunidade de fazer isso. Basicamente, ela está trabalhando como um programador em C com outros membros da equipe. Ela admite que o trabalho é desafiador, mas está consciente de que não está desenvolvendo suas habilidades com interfaces de hardware. Está preocupada com a dificuldade em encontrar um trabalho que envolva interface de hardware após esse projeto. Como ela não quer perturbar a equipe revelando que está pensando no próximo projeto, ela decidiu diminuir a conversa com eles.

Nesse exemplo, Alice tenta descobrir se as circunstâncias pessoais de Dorothy podem ser o problema. As dificuldades pessoais normalmente afetam a motivação, pois a pessoa não se concentra em seu trabalho. Talvez você deva dar tempo e apoio às pessoas para resolver essas questões, embora também tenha de deixar claro que as pessoas envolvidas ainda têm responsabilidade com seu empregador.

De fato, o problema de motivação de Dorothy é um daqueles que surgem muito freqüentemente quando os projetos se desenvolvem em uma direção não prevista. As pessoas que esperam fazer um tipo de trabalho podem acabar fazendo algo completamente diferente. Isso se torna um problema quando as pessoas querem desenvolver suas habilidade de algum modo diferente do conduzido pelo projeto. Nessas circunstâncias, você pode decidir que o membro deve deixar a equipe e procurar oportunidades em algum outro lugar. No exemplo, contudo, Alice decide tentar convencer Dorothy que ampliar sua experiência é um passo positivo na carreira. Ela dá a Dorothy mais autonomia no projeto e organiza cursos de treinamento em engenharia de software que oferecerão a ela muitas oportunidades depois que o projeto tiver terminado.

O principal problema com o modelo de motivação de Maslow é que ele assume um ponto de vista exclusivamente pessoal sobre a motivação. Ele não leva em consideração o fato de que as pessoas se sentem parte de uma organização, de um grupo profissional e, geralmente, de alguma cultura. Essa não é simplesmente uma questão de satisfação de necessidades sociais — as pessoas podem ser motivadas em ajudar o grupo a atingir suas metas.

Ser membro de um grupo coeso é altamente interessante para a maioria das pessoas. Pessoas com trabalhos gratificantes geralmente gostam de ir ao trabalho pois estão motivadas pelas pessoas com quem trabalham e pelo trabalho que realizam. Portanto, assim como pensar em motivação de pessoas, você deve também pensar sobre como o grupo como um todo pode ser motivado para atingir as metas da organização. Explico as questões de gerenciamento de grupos na próxima seção.

Em um estudo psicológico sobre motivação, Bass e Dunteman (Bass e Dunteman, 1963) classificaram os profissionais em três tipos:

1. *Orientado a tarefas*. Pessoas motivadas pelo trabalho que realizam. Em engenharia de software, eles são técnicos motivados pelo desafio intelectual do desenvolvimento de software.
2. *Auto-orientados*. Pessoas motivadas principalmente pelo sucesso e reconhecimento pessoal. Eles estão interessados no desenvolvimento do software como uma maneira de atingir suas próprias metas. Isso não significa que essas pessoas sejam egoístas ou pensem somente em seus próprios assuntos. Em vez disso, eles freqüentemente têm metas de longo prazo, como progressão na carreira, que os motivam. Desejam ser bem-sucedidos em seu trabalho para ajudar a atingir essas metas.
3. *Orientado a interações*. Pessoas motivadas pela presença e pelas ações de seus colegas de trabalho. À medida que o desenvolvimento de software torna-se mais centrado no usuário, mais pessoas orientadas a interações estão se envolvendo na engenharia de software.

As personalidades orientadas a interações, em geral, gostam de trabalhar em grupo, enquanto as pessoas orientadas a tarefas e auto-orientadas preferem, muitas vezes, trabalhar sozinhas. As mulheres são mais propensas a serem orientadas a interações do que os homens. Elas freqüentemente são mais comunicativas. Explico a combinação desses tipos diferentes de personalidade em grupo na Seção 25.3.2.

A motivação de cada indivíduo é constituída de elementos de cada classe, mas geralmente um tipo de motivação é dominante em qualquer momento. Contudo, as personalidades não são estáticas e as pessoas podem mudar. Por exemplo, pessoas técnicas que sentem que não estão sendo apropriadamente recompensadas podem se tornar auto-orientadas e colocar seus interesses pessoais antes dos assuntos técnicos.

25.3 Gerenciamento de grupos

A maioria do software profissional é desenvolvida por equipes de projeto que variam em tamanho de duas a várias centenas de pessoas. Contudo, como é obviamente impossível para essas pessoas trabalharem juntas em um único problema, essas grandes equipes são, em geral, divididas em vários grupos. Cada grupo é responsável por uma parte do sistema total. Como regra geral, os grupos de projeto de engenharia de software não devem ter, normalmente, mais do que oito ou dez membros. Quando pequenos grupos são usados, os problemas de comunicação são reduzidos. O grupo inteiro pode se reunir em torno de uma mesa e em seus próprios escritórios.

Colocar junto um grupo que trabalhe eficazmente é, portanto, uma tarefa crítica de gerenciamento. Obviamente é importante que o grupo tenha o equilíbrio certo de habilidades técnicas, experiências e personalidades. Contudo, grupos bem-sucedidos são mais que simplesmente um conjunto de indivíduos com equilíbrio certo de habilidades. Um bom grupo tem um espírito de equipe, de modo que as pessoas envolvidas estejam motivadas pelo sucesso do grupo, assim como por suas próprias metas pessoais.

Existe uma série de fatores que influenciam o trabalho em grupo:

1. *Composição do grupo*. Existe o equilíbrio certo entre as habilidades, experiências e personalidades na equipe?
2. *Coesão do grupo*. O grupo pensa em si como uma equipe, em vez de como um conjunto de indivíduos que trabalham juntos?
3. *Comunicação do grupo*. Os membros do grupo se comunicam eficazmente uns com os outros?
4. *Organização do grupo*. O grupo está organizado de maneira que cada pessoa se sinta valorizada e satisfeita com seu papel no grupo?

25.3.1 Composição do grupo

Conforme expliquei na Seção 25.2, muitos engenheiros de software são motivados principalmente por seu trabalho. Grupos de desenvolvimento de software, portanto, são compostos muitas vezes de pessoas que têm suas próprias idéias sobre como os problemas técnicos devem ser resolvidos. Isso é confirmado pelos problemas regularmente relatados a respeito de padrões de interface ignorados, sistemas que são reprojetados enquanto estão codificados, recursos de sistema desnecessários etc. Você deve tentar, se possível, selecionar membros de modo a evitar esses problemas.

Um grupo com personalidades complementares pode trabalhar melhor que um grupo selecionado somente com base em habilidades técnicas. As pessoas motivadas pelo trabalho são, provavelmente, os mais fortes tecnicamente. As pessoas auto-orientadas, provavelmente, serão as melhores para impulsionar o trabalho adiante em direção a seu término. As pessoas orientadas a interações ajudam a facilitar as comunicações dentro do grupo. Penso que é particularmente importante ter pessoas orientadas a interações em um grupo. Eles gostam de conversar com pessoas e podem detectar tensões e desacordos em um estágio inicial, antes que tenham sérios impactos no grupo.

No exemplo de estudo de caso mostrado no Quadro 25.3, mostro como Alice, a gerente do projeto, tentou criar um grupo com personalidades complementares. Esse grupo específico tem uma boa combinação de pessoas orientadas a interações e orientadas a tarefas, mas já expliquei no Quadro 25.2 como a personalidade auto-orientada de Dorothy pode causar problemas. A função de meio período de Fred no grupo, como especialista de domínio, também pode ser um problema. Ele está muito mais interessado em desafios técnicos e, assim, pode não interagir bem com outros membros do grupo. O fato de ele nem sempre ser parte da equipe pode desviá-lo das metas do grupo.

Algumas vezes, é impossível escolher um grupo com personalidades complementares. Nesse caso, o gerente de projeto precisa controlar o grupo de modo que as metas individuais não transcendam os objetivos organizacionais e do grupo. Esse controle será mais fácil se todos os membros do grupo participarem de cada estágio do projeto. Há grande chance de que ocorram iniciativas individuais quando são dadas instruções aos membros do grupo sem que eles estejam cientes do papel que desempenham no projeto total.

Por exemplo, digamos que tenha sido dado a um engenheiro um projeto de programa para codificação e ele observa possíveis melhorias no projeto. Se ele implementar essas melhorias sem compreender as razões do projeto original, elas podem ter implicações adversas em outras partes do sistema. Se todos os membros do grupo estiverem envolvidos no projeto desde seu início, eles compreenderão por que as decisões de projeto foram tomadas. Eles podem se identificar com essas decisões em vez de se oporem a elas.

O líder do grupo tem importante papel. Ele pode ser responsável pela direção técnica e administração do projeto. Os líderes de grupo devem manter o acompanhamento diário do trabalho de seu grupo, assegurar que as pessoas estão trabalhando de maneira eficaz e trabalhar estreitamente com os gerentes no planejamento de projeto.

Quadro 25.3

Composição do grupo.

Estudo de caso 3: Composição do grupo

Na criação de um grupo para desenvolvimento de tecnologia de assistência, Alice está consciente da importância da seleção de membros com personalidades complementares. Nas entrevistas pessoais, ela tenta avaliar se as pessoas são orientadas a tarefas, auto-orientadas ou orientadas a interações. Ela acredita ser predominantemente do tipo auto-orientada e acredita que nesse projeto será observada pela gerência-sênior e será promovida. Ela, portanto, procura por uma ou talvez duas personalidades orientadas a interações e quer pessoas orientadas a tarefas para completar a equipe. Na avaliação final, ela chega a:

Alice — Auto-orientada
Brian — Orientado a tarefas
Bob — Orientado a tarefas
Carol — Orientada a interações
Dorothy — Auto-orientada
Ed — Orientado a interações
Fred — Orientado a tarefas

Os líderes são, normalmente, indicados e se reportam ao gerente geral do projeto. Contudo, o líder indicado pode não ser um líder real do grupo em relação a assuntos técnicos. Os membros do grupo podem procurar um outro membro para liderá-los. Ele pode ser um engenheiro mais competente tecnicamente ou pode ser mais motivador que o líder de grupo indicado.

Algumas vezes, é interessante separar a liderança técnica da administração do projeto. Pessoas tecnicamente competentes nem sempre são os melhores administradores. Ao receberem uma função administrativa, seu valor perante o grupo pode ser diminuído. É melhor apoiar o grupo com um administrador que possa aliviá-los das tarefas do dia-a-dia.

A imposição de um líder inadequado ao grupo provavelmente causará tensões. Os membros da equipe não respeitarão o líder e poderão rejeitar a lealdade do grupo em favor de metas individuais. Esse é um problema específico em uma área caracterizada por rápidas mudanças, como é a engenharia de software, na qual novos membros podem estar mais atualizados e mais bem formados do que os líderes de grupo experientes. Algumas pessoas experientes podem se ressentir com a imposição de um líder jovem com novas idéias.

25.3.2 Coesão do grupo

Em um grupo coeso, seus membros pensam no grupo como mais importante do que as pessoas que pertencem a ele. Os membros de um grupo bem liderado, coeso, são leais a ele. Identificam-se com as metas do grupo e com os seus membros. Eles tentam proteger o grupo, como uma entidade, em relação às interferências externas. Isso torna o grupo robusto e capaz de lidar com problemas e situações inesperadas. O grupo pode lidar com mudanças fornecendo apoio e ajuda mútuos.

As vantagens de um grupo coeso são:

1. *Pode ser desenvolvido um padrão de qualidade de grupo*. Devido a esse padrão ser estabelecido por consenso, é mais provável que seja seguido do que os padrões externos impostos ao grupo.
2. *Os membros do grupo trabalham rigorosamente em conjunto*. As pessoas no grupo aprendem umas com as outras. As inibições causadas por ignorância são minimizadas uma vez que o aprendizado mútuo é encorajado.
3. *Os membros do grupo podem conhecer o trabalho uns dos outros*. A continuidade poderá ser mantida caso um membro deixe o grupo. Os outros membros podem assumir tarefas críticas e assegurar que o projeto não seja interrompido impropriamente.
4. *A programação sem egos exaltados pode ser praticada*. Os programas são considerados como propriedade do grupo, e não uma propriedade pessoal.

A programação sem egos exaltados (Weinberg, 1971) é um estilo de trabalho em grupo no qual projetos, programas e outros documentos são considerados de propriedade comum do grupo em vez de das pessoas que os escreveram. Na cultura de programação sem egos exaltados, é mais provável que as pessoas ofereçam seu trabalho para inspeção por outros membros do grupo, aceitem críticas e trabalhem em grupo para aprimorar o programa. A coesão de grupo aumenta porque todos os membros têm uma responsabilidade conjunta sobre o software. A idéia de programação sem egos exaltados é fundamental para a extreme programming (Beck, 2000), explicada no Capítulo 17. O aprimoramento constante do código do sistema, independentemente de quem escreveu o código, é um dos princípios básicos da extreme programming.

Assim como aprimorar a qualidade dos projetos, programas e documentos, a programação sem egos exaltados aprimora também a comunicação dentro do grupo. Ela encoraja discussões desinibidas sem considerações de status, experiência ou sexo. Os membros cooperam ativamente com outros membros do grupo durante o andamento do projeto. Isso torna os membros de um grupo unidos e faz com que eles se sintam parte da equipe.

A coesão do grupo depende de muitos fatores, entre eles a cultura organizacional e as personalidades no grupo. Os gerentes podem encorajar a coesão de várias maneiras: podem organizar eventos sociais para os membros do grupo e suas famílias, podem tentar estabelecer um sentido de identidade do grupo dando a ele um nome e definindo uma identidade e um território, ou podem envolver os membros em atividades praticadas em grupo, como esportes e jogos.

Uma das maneiras mais eficientes de promover a coesão é incluir-se e se assegurar de que os membros do grupo sejam tratados como pessoas responsáveis e de confiança, e conceder-lhes acesso a informações. Freqüentemente, os gerentes sentem que não podem revelar certas informações a todos do grupo. Isso cria, invariavelmente, um clima de desconfiança. Uma simples troca de informações é uma maneira barata e eficiente de fazer com que as pessoas se sintam parte de uma equipe.

Veremos um exemplo desse fragmento de estudo de caso no Quadro 25.4. Alice organiza reuniões informais regulares nas quais ela conta aos demais membros do grupo o que está acontecendo. Ela estabelece um ponto de discussão que envolva as pessoas no desenvolvimento do produto, pedindo-lhes que proponham novas idéias derivadas de suas próprias experiências familiares. Os eventos 'um dia fora' são uma boa maneira de promover a coesão — as pessoas relaxam enquanto ajudam umas às outras a aprender sobre novas tecnologias.

Quadro 25.4

Coesão do grupo.

Estudo de caso 4: Espírito de equipe

Alice, gerente de projeto experiente, compreende a importância da criação de um grupo coeso. Como o desenvolvimento é de um novo produto, ela aproveita a oportunidade para envolver todos os membros do grupo na especificação e projeto do produto incentivando-os a discutir possíveis tecnologias com membros idosos de suas famílias e a trazê-los para o almoço semanal com o grupo. O almoço com o grupo é uma oportunidade para que todos os membros da equipe se encontrem informalmente, conversem sobre questões relacionadas e conheçam uns aos outros.

No almoço, Alice conta aos membros da equipe o que ela sabe sobre as novidades organizacionais, políticas, estratégicas etc. Cada membro da equipe então resume o que esteve fazendo e o grupo discute um tópico geral, como novas idéias de produto provenientes de familiares idosos.

A cada poucos meses, Alice organiza 'um dia fora' para o grupo, no qual a equipe passa dois dias em 'atualização tecnológica'. Cada membro da equipe prepara um relatório atualizado sobre uma tecnologia relevante e a apresenta ao grupo. Essa é uma reunião fora do escritório em um bom hotel e com bastante tempo para discussões e interações sociais.

Contudo, grupos coesos e fortes podem, às vezes, sofrer de dois problemas:

1. *Resistência irracional a uma mudança de liderança*. Se o líder de um grupo coeso tiver de ser substituído por alguém de fora do grupo, os membros podem se unir contra o novo líder. Os membros do grupo podem passar bastante tempo resistindo às mudanças propostas pelo novo líder, levando a uma conseqüente perda de produtividade. Sempre que possível, os novos líderes deverão ser indicados de dentro do grupo.
2. *Pensamento em grupo (groupthink)*. O pensamento em grupo (Janis, 1972) acontece quando as habilidades críticas dos membros estão desgastadas pela lealdade ao grupo. A consideração de alternativas é substituída pela lealdade às normas e às decisões do grupo. Qualquer proposta aceita pela maioria do grupo pode ser adotada sem consideração apropriada de alternativas.

Para evitar o pensamento em grupo, você deve organizar sessões formais nas quais os membros do grupo sejam encorajados a questionar as decisões tomadas. Especialistas externos podem participar para revisar as decisões do grupo. Pessoas naturalmente argumentativas, questionadoras e desrespeitosas em relação ao *status quo* podem ser nomeadas como líderes das discussões. Eles atuam como 'advogado do diabo', questionando constantemente as decisões do grupo e, portanto, forçando os membros a pensar sobre o assunto e avaliar suas atividades.

25.3.3 Comunicação do grupo

A boa comunicação entre os membros de um grupo de desenvolvimento de software é essencial. Os membros do grupo devem trocar informações sobre o status de seu trabalho, as decisões de projeto tomadas e as mudanças necessárias em decisões anteriores. A boa comunicação também fortalece a coesão do grupo uma vez que os membros passam a compreender as motivações, os pontos fortes e fracos de outras pessoas no grupo.

Alguns fatores principais que influenciam a eficiência da comunicação são:

1. *Tamanho do grupo*. À medida que um grupo cresce em tamanho, torna-se mais difícil assegurar que todos os membros se comuniquem eficientemente uns com os outros. O número de elos de comunicação em uma direção é $n \cdot (n - 1)$, sendo n o tamanho do grupo; portanto, em um grupo de sete a oito membros, é muito provável que algumas pessoas raramente se comuniquem. As diferenças de status entre os membros do grupo significam que as comunicações geralmente ocorrem em uma direção. Os membros com status superior tendem a dominar as comunicações com os membros de status inferior, que são, muitas vezes, relutantes em iniciar uma conversa ou em fazer observações críticas.
2. *Estrutura do grupo*. As pessoas em grupos estruturados informalmente se comunicam com mais eficiência do que as pessoas em grupos com uma estrutura formal e hierárquica. Em grupos hierárquicos, as comunicações tendem a fluir para cima e para baixo pela hierarquia. As pessoas de mesmo nível podem não conversar umas com as outras. Esse é um problema específico em um projeto de grande porte com vários grupos de desenvolvimento. Quando as pessoas que trabalham em subsistemas diferentes se comunicam somente por meio de seus gerentes, o projeto pode sofrer atrasos e haver desentendimentos.
3. *Composição do grupo*. Pessoas com o mesmo tipo de personalidade podem entrar em conflito e a comunicação pode ser comprometida. A comunicação geralmente é melhor em grupos compostos de homens e mulheres (Marshall e Heslin, 1975) do que em grupos com pessoas de mesmo sexo. As mulheres tendem a ser mais orientadas a interações do que os homens e podem atuar como controladoras e facilitadoras das interações do grupo.

4. *Ambiente físico de trabalho*. A organização do local de trabalho é um fator importante para facilitar ou inibir as comunicações. Explico isso mais adiante na Seção 25.3.5.

25.3.4 Organização do grupo

Pequenos grupos de programação geralmente são organizados de maneira bastante informal. O líder do grupo se envolve no desenvolvimento do software com os demais membros. Um líder técnico pode surgir como aquele que efetivamente controla a produção de software. Em um grupo informal, o trabalho a ser realizado é discutido pelo grupo como um todo e as tarefas são alocadas de acordo com a habilidade e a experiência. Os membros mais experientes do grupo podem ser responsáveis pelo projeto de arquitetura. Contudo, o projeto detalhado e a implementação são de responsabilidade do membro da equipe alocado a uma tarefa específica.

Grupos informais podem ser bem-sucedidos, particularmente quando a maioria dos membros do grupo é experiente e competente. Como o grupo toma decisões por consenso, isso melhora o espírito, a coesão e o desempenho do grupo. Se um grupo for composto por membros em sua maioria inexperientes ou incompetentes, a informalidade pode ser um obstáculo, pois não existe uma autoridade definida para dirigir os trabalhos, provocando falta de coordenação entre os membros do grupo e, possivelmente, eventuais falhas de projeto.

Uma variante de organização de grupo democrático interessante é descrita por Beck em seu livro sobre *extreme programming* (Beck, 2000). Nessa abordagem, muitas decisões geralmente vistas como decisões gerenciais, como decisões de cronograma, são repassadas aos membros do grupo. Os programadores trabalham juntos, em pares, para desenvolver o código e assumem juntos a responsabilidade pelos programas desenvolvidos.

Conforme explicado no Capítulo 26, a habilidade individual transparece mais na produtividade do programador. Os melhores programadores podem ser até 25 vezes mais produtivos que os piores programadores. Então, faz sentido usar as melhores pessoas de maneira mais eficiente e dar a elas todo o apoio possível.

Para fazer uso mais eficiente de programadores muito habilidosos, Baker (Baker, 1972) e outros (Aron, 1974; Brooks, 1975) sugerem que as equipes devem ser formadas em torno de um programador-chefe muito habilidoso. O princípio básico da equipe de programador-chefe é que pessoas habilidosas e experientes devem ser responsáveis por todo o desenvolvimento do software. Elas não devem se preocupar com assuntos rotineiros e devem ter um bom apoio técnico e administrativo para seu trabalho. Devem enfocar o software a ser desenvolvido, e não se envolver em reuniões externas.

No entanto, a organização da equipe de programador-chefe tem sérios problemas porque é muito dependente deste e de seus assistentes. Outros membros da equipe, que não recebem responsabilidade suficiente, tornam-se desmotivados porque sentem que suas habilidades estão sendo subutilizadas.

Contudo, o princípio geral de incluir especialistas na equipe de programação é bom. Quando os membros da equipe são escolhidos, você pode enfocar pessoas com habilidades genéricas, como a capacidade de comunicação e de resolução de problemas e, então, trazer especialistas conforme for necessário durante o projeto. O uso de especialistas de acordo com as necessidades também significa que desenvolvedores relativamente inexperientes terão oportunidade de aprender e desenvolver suas habilidades à medida que o projeto progredir.

25.3.5 Ambientes de trabalho

O local de trabalho tem efeito importante sobre o desempenho das pessoas e sobre a satisfação de seu trabalho. Experimentos psicológicos têm mostrado que o comportamento é afetado pelo tamanho da sala, pela mobília, pelos equipamentos, pela temperatura, umidade, brilho e qualidade da luz, pelo barulho e o grau de privacidade disponível. O comportamento do grupo é afetado pela organização arquitetônica e recursos de telecomunicações. As comunicações dentro de um grupo são afetadas pela arquitetura do prédio e pela estrutura do espaço de trabalho.

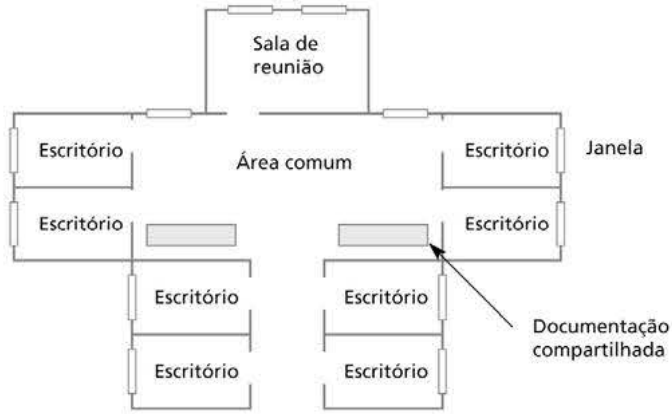
Existe um custo real e significativo na falha de fornecimento de boas condições de trabalho. Quando as pessoas estão infelizes com suas condições de trabalho, aumenta a rotatividade de pessoal. Portanto, mais custos devem ser despendidos em recrutamento e treinamento. Projetos de software podem ser atrasados por falta de pessoal qualificado (DeMarco e Lister, 1999).

O pessoal de desenvolvimento de software trabalha, muitas vezes, em grandes áreas planas e abertas e, algumas vezes, em saletas, sendo que somente a gerência sênior tem escritórios individuais. McCue (McCue, 1978) realizou um estudo que mostrava que a arquitetura plana e aberta fornecida por muitas organizações não era popular nem produtiva. Os fatores mais importantes identificados nesse estudo de projeto foram:

1. *Privacidade*. Os programadores exigem uma área em que possam se concentrar e trabalhar, sem interrupções.
2. *Consciência externa*. As pessoas preferem trabalhar com luz natural e com uma visão do ambiente externo.
3. *Personalização*. Os indivíduos adotam diferentes práticas de trabalho e têm opiniões diferentes sobre a decoração. A habilidade de reorganizar o espaço para se adequar às práticas de trabalho e personalizar esse ambiente é importante.

Figura 25.2

Agrupamento de escritórios e sala de reunião.



Em resumo, as pessoas gostam de escritórios individuais que possam organizar de acordo com seus gostos e necessidades. Espaços individuais sofrem menos perturbações e interrupções do que espaços planos e abertos. Nos escritórios planos e abertos, é negado às pessoas um ambiente de trabalho calmo e privativo. Esses escritórios são limitados quanto ao modo como as pessoas podem personalizar seu espaço de trabalho. A concentração pode ser difícil e o desempenho, degradado.

O fornecimento de escritórios individuais para o pessoal de engenharia de software pode fazer uma diferença significativa na produtividade. DeMarco e Lister (DeMarco e Lister, 1985) compararam a produtividade de programadores em diferentes tipos de espaços de trabalho. Eles descobriram que fatores como o espaço de trabalho privativo e a capacidade de evitar interrupções têm efeito significativo. Os programadores com boas condições de trabalho eram duas vezes mais produtivos do que os programadores igualmente habilidosos que tinham de trabalhar em condições mais precárias.

Os grupos de desenvolvimento necessitam de áreas onde todos os membros do grupo possam se reunir e discutir seus projetos, tanto formal como informalmente. As salas de reunião devem ser capazes de acomodar o grupo inteiro com privacidade. Os requisitos de privacidade individual e de comunicação em grupo parecem ser objetivos exclusivos. McCue sugeriu que o agrupamento de escritórios individuais em torno de salas de reuniões de grupo maiores (Figura 25.2) é a melhor maneira de conciliar esses requisitos conflitantes.

Um modelo similar foi sugerido por Beck em sua descrição sobre o ambiente para *extreme programming*. Contudo, ele sugere que seja mantida uma área plana e aberta com todas as atividades de programação sendo realizadas na área comum e, quando as pessoas desejarem trabalhar sozinhas, em saletas. Claramente, o requisito principal é fornecer espaço tanto individual como de grupo de modo que as pessoas possam trabalhar sozinhas ou em grupo, quando necessário.

Os recursos de comunicação, por meio de áreas de reuniões, ajudam as pessoas a resolverem seus problemas e a trocarem informações de maneira informal, porém eficiente. Weinberg (Weinberg, 1971) cita um exemplo engraçado de uma organização que desejava acabar com o desperdício de tempo dos programadores que conversavam em volta de uma máquina de café. A máquina foi removida e, em seguida, houve imediatamente um drástico aumento de solicitações para apoio de programação formal. Além de conversarem em volta da máquina, as pessoas estavam resolvendo problemas uns dos outros. Isso mostra como as empresas necessitam de locais para reuniões informais, além das salas formais de reunião.

O estudo de caso do Quadro 25.5 ilustra que, muitas vezes, é necessário trabalhar dentro das restrições dos edifícios. Você não pode adaptá-los ou ter tantos espaços quanto gostaria. Nesse exemplo, Alice usou um único escritório como

Quadro 25.5

Organização do escritório.

Estudo de caso 5: Organização do escritório

Alice compreende a importância dos ambientes de trabalho, mas sua empresa ocupa um prédio da década de 1970 que não pode ser adaptado para uma estrutura ideal. Três escritórios foram designados para a sua equipe — um pequeno, separado, e dois adjacentes maiores, cada um com quatro mesas. Dois membros da equipe (Carol e Brian) trabalham muitas vezes em casa e Fred, o especialista em alarmes, trabalha com a equipe somente duas vezes por semana. A equipe tem acesso a uma sala de reunião compartilhada por outros grupos e cada andar do prédio tem uma sala de café para interações informais.

Em vez de usar a pequena sala como seu escritório pessoal, conforme destinado pela gerência, Alice decide que esse deve ser um espaço tranqüilo de 'concentração' que pode ser usado por qualquer membro da equipe que necessite trabalhar sem distração. Ela escolhe uma das salas como um escritório de desenvolvimento com mesas para protótipos de interfaces com o usuário em hardware e papel. Essa sala tem também uma mesa usada na maioria das vezes por Fred, quando ele está trabalhando com a equipe, mas é também compartilhada com Carol e Brian quando eles trabalham no escritório. Alice compartilha o outro escritório com Bob, Dorothy e Ed. O prédio tem rede sem fio e todos os membros da equipe têm laptops.

espaço para trabalho concentrado e para trabalho de desenvolvimento confinado, no qual as pessoas estavam discutindo o que fazer dentro de uma única sala. As mesas compartilhadas são cada vez mais comuns para membros de equipe que nem sempre estão trabalhando no escritório. Como cada membro da equipe tem um laptop, eles podem trabalhar em qualquer lugar — em suas mesas, nas calmas salas ou nos espaços sociais compartilhados, dentro do prédio.

25.4 O Modelo de Maturidade de Capacitação de Pessoal

O Software Engineering Institute (SEI) nos Estados Unidos está engajado em um programa de longo prazo de aprimoramento do processo de software. Parte desse programa é o Modelo de Maturidade de Capacitação (CMM — *Capability Maturity Model*) para processos de software, o qual explico no Capítulo 28. Ele está relacionado com as melhores práticas em engenharia de software. Para apoiar esse modelo, o instituto também propôs o Modelo de Maturidade de Capacitação de Pessoal (P-CMM — *People Capability Maturity Model*) (Curtis, et al., 2001). O P-CMM pode ser usado como framework de aprimoramento da maneira pela qual uma organização gerencia seu patrimônio humano.

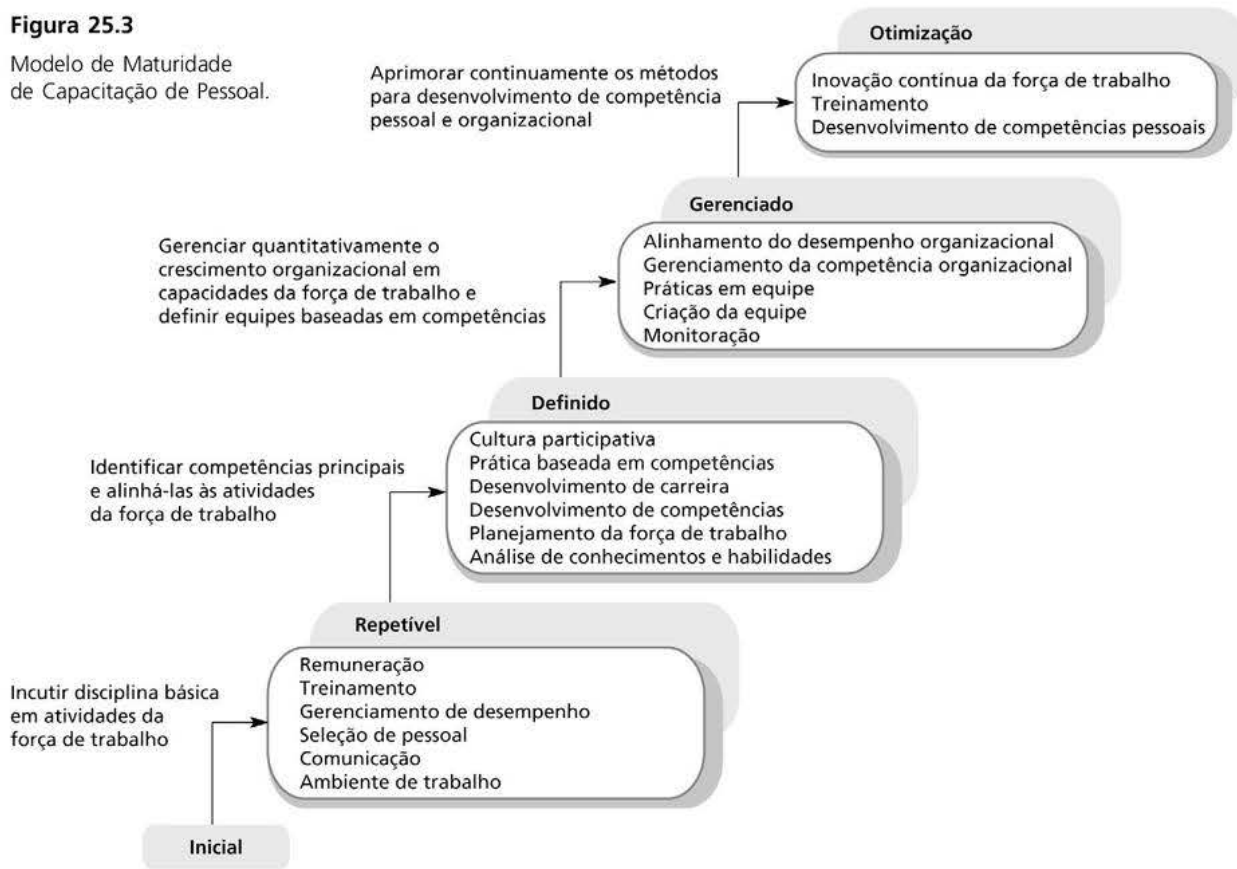
Assim como o CMM, o P-CMM é um modelo de cinco níveis, conforme mostrado na Figura 25.3. Os cinco níveis são:

1. *Inicial*. Práticas gerenciais informais específicas de pessoal.
2. *Repetível*. Definição de políticas para desenvolvimento de capacitação de pessoal.
3. *Definido*. Padronização das melhores práticas de gerenciamento de pessoal na organização.
4. *Gerenciado*. Metas quantitativas para o gerenciamento de pessoal.
5. *Otimização*. Foco contínuo no aprimoramento da competência individual e motivação da força de trabalho.

Curtis et al. (Curtis, et al., 2001) estabelecem que os objetivos estratégicos do P-CMM são:

1. Aprimorar a capacitação das organizações de software pelo aumento da capacitação de sua força de trabalho.
2. Assegurar que a capacitação do desenvolvimento de software seja um atributo da organização em vez de apenas de alguns indivíduos.

Figura 25.3
Modelo de Maturidade de Capacitação de Pessoal.



3. Alinhar a motivação dos indivíduos com a da organização.
4. Reter o patrimônio humano (isto é, pessoas com conhecimento e habilidades críticas) na organização.

O P-CMM é uma ferramenta prática para aprimoramento do gerenciamento de pessoal em uma organização pois fornece um framework para motivação, reconhecimento, padronização e aprimoramento de boas práticas. Entretanto, como todos os modelos de capacitação criados pelo SEI, ele é projetado para grandes empresas em vez de para pequenas empresas. Ele reforça a necessidade de reconhecer a importância das pessoas como indivíduos e de desenvolver suas capacidades. Naturalmente, a aplicação completa desse modelo é muito dispendiosa e, provavelmente, desnecessária para a maioria das organizações. Contudo, ele é um guia útil que pode conduzir a melhorias significativas na capacidade da organização em produzir software de alta qualidade.

PONTOS-CHAVE

- A seleção de pessoal para trabalhar em um projeto é uma tarefa importante para os gerentes de projeto. Os fatores que poderiam ser usados para selecionar o pessoal incluem experiência em domínio de aplicação, adaptabilidade e personalidade.
- As pessoas são motivadas pelas interações com outras pessoas, pelo reconhecimento da gerência e de seus pares e pelas oportunidades dadas para o desenvolvimento pessoal.
- Os grupos de desenvolvimento de software devem ser pequenos e coesos. Os líderes de grupo devem ser tecnicamente competentes e devem ter apoio administrativo e técnico.
- A comunicação dentro de um grupo é influenciada por fatores como o status dos membros do grupo, o tamanho do grupo, a distribuição de sexos do grupo, as personalidades e os canais de comunicação disponíveis.
- Os ambientes de trabalho das equipes devem incluir espaços nos quais a equipe possa interagir e os membros da equipe possam se concentrar rapidamente em seu trabalho.
- O Modelo de Maturidade de Capacitação de Pessoal (P-CMM) fornece framework e recomendações associadas para o aprimoramento da capacitação do pessoal e da capacitação da organização em obter benefícios de seu patrimônio humano.

LEITURAS SUGERIDAS

A handbook of software and systems engineering: empirical observations, laws and theories. Este livro dedica-se às descobertas empíricas, hipóteses e teorias relevantes para engenharia de software. O Capítulo 10 aborda habilidades, motivações e satisfação dos usuários e discute teorias de psicologia que apóiam o material deste capítulo (A. Andres e D. Rombach, 2003, Addison-Wesley.)

Software management, 6th ed. Este é um texto tutorial do IEEE com diversos artigos sobre gerenciamento e motivação de pessoal (D. J. Reifer, 2002, Wiley-IEEE Press.)

The people capability maturity model: guidelines for improving the workforce. Este livro é uma descrição completa do P-CMM, incluindo um guia de aprimoramento da capacitação individual, do desenvolvimento de uma forte cultura organizacional, da medição do desempenho e da criação de uma força de trabalho flexível (B. Curtis, et al., 2001, Addison-Wesley.)

Peopleware: productive projects and teams, 2nd ed. Este é um livro clássico sobre a importância de tratar as pessoas adequadamente ao gerenciar projetos de software. É fácil de ler e é um dos poucos livros que reconhece a importância do local onde as pessoas trabalham. Fortemente recomendado (T. DeMarco e T. Lister, 1999, Dorset House.)

EXERCÍCIOS

25.1 Explique por que a consistência, o respeito, a inclusão e a honestidade são fatores que contribuem para um gerenciamento eficiente de pessoal.

25.2 Quais fatores devem ser levados em consideração na seleção de pessoal para trabalhar em um projeto de desenvolvimento de software? Fornecendo razões para sua resposta, sugira quais dessas razões seriam as mais importantes na escolha do pessoal para um projeto de desenvolvimento de sistemas embarcados de tempo real de um controlador para uma máquina de cirurgia de olhos.

25.3 Desenvolva um exemplo de estudo de caso sobre motivação como o do Quadro 25.2 para incluir atividades gerais que Alice poderia introduzir a fim de assegurar que os outros membros da equipe permaneçam motivados.

25.4 Explique por que manter todos os membros de um grupo informados sobre o progresso e as decisões técnicas em um projeto pode aumentar a coesão do grupo.

25.5 Explique o que você entende por *groupthink*. Descreva os perigos desse fenômeno e explique como isso pode ser evitado.

25.6 Quais problemas você imagina que podem surgir em equipes de *extreme programming* na qual muitas decisões gerenciais são repassadas aos próprios membros da equipe?

25.7 Por que os escritórios comuns planos e abertos são, às vezes, menos adequados para o desenvolvimento de software do que escritórios individuais? Sob quais circunstâncias você imagina que os ambientes planos e abertos podem ser melhores?

25.8 Por que o P-CMM é um framework eficaz para o aprimoramento do gerenciamento de pessoal em uma organização? Sugira como ele pode ser modificado caso seja usado em pequenas empresas.

25.9 Os gerentes devem ser amigáveis e sociáveis com os membros menos experientes de seu grupo?

25.10 É ético fornecer as respostas que você imagina que o interlocutor quer ouvir em vez de dizer o que você realmente sente quando se submete a testes psicológicos e de aptidão?

26

Estimativa de custo de software

Objetivos

O objetivo deste capítulo é apresentar técnicas de estimativa de custo e esforço necessários para a produção de software. Após ler este capítulo, você:

- compreenderá os fundamentos de custos de software e as razões pelas quais o preço do software pode não estar diretamente relacionado a seu custo de desenvolvimento;
- terá sido apresentado a três métricas usadas para a avaliação de produtividade de software;
- avaliará por que várias técnicas devem ser usadas na estimativa de custos e prazos de software;
- compreenderá os princípios do modelo COCOMO II para a estimativa algorítmica de custo.

Conteúdo

26.1 Produtividade de software
26.2 Técnicas de estimativa
26.3 Modelagem algorítmica de custos
26.4 Duração de projeto e seleção de pessoal

No Capítulo 5, apresentei o processo de planejamento de projeto, no qual o trabalho em um projeto é dividido em uma série de atividades separadas. Nessa explicação anterior de planejamento de projeto, concentrei-me nas formas de representação dessas atividades, suas dependências e a alocação de pessoas para realizar essas tarefas. Neste capítulo, volto ao problema de associar estimativas de esforço e tempo com as atividades de projeto. A estimativa envolve respostas às questões:

1. Quanto esforço é necessário para completar cada atividade?
2. Quanto tempo de calendário é necessário para completar cada atividade?
3. Qual é o custo total de cada atividade?

As estimativas de custo e de prazo de projeto são normalmente realizadas em conjunto. Os custos de desenvolvimento são primariamente os custos de esforço envolvido, de modo que o cálculo de esforço é usado para ambas as estimativas, de custo e de cronograma. Contudo, você

Tabela 26.1 Fatores que afetam a atribuição de preço de software

Fator	Descrição
Oportunidade de mercado	Uma organização de desenvolvimento pode cotar um preço baixo porque deseja entrar em um novo segmento de mercado de software. A aceitação de um lucro pequeno sobre um projeto pode dar à organização a oportunidade de obter um lucro maior posteriormente. A experiência obtida pode ajudar no desenvolvimento de novos produtos.
Incerteza de estimativa de custo	Se uma empresa está insegura sobre sua estimativa de custo, ela pode aumentar o preço com lucro bem acima do normal devido a alguma contingência.
Termos contratuais	Um cliente pode estar disposto a permitir que o desenvolvedor mantenha a propriedade sobre os códigos-fonte e reusá-lo em outros projetos. O preço cobrado pode, então, ser menor do que se os códigos-fonte do software fossem entregues ao cliente.
Volatilidade de requisitos	Se os requisitos provavelmente vão mudar, uma organização pode diminuir seu preço para ganhar um contrato. Depois que o contrato for concedido, poderão ser cobrados preços altos por mudanças nos requisitos.
Saúde financeira	Os desenvolvedores em dificuldades financeiras podem diminuir seus preços para ganhar um contrato. É melhor obter um lucro menor do que o normal ou mesmo apenas recuperar os custos do que ficar fora do negócio.

pode fazer uma estimativa de custo antes que os cronogramas detalhados sejam elaborados. Essas estimativas iniciais podem ser usadas para estabelecer um orçamento para o projeto ou para estabelecer o preço do software para um cliente.

Existem três parâmetros envolvidos no cálculo do custo total de um projeto de desenvolvimento de software:

- Custos de hardware e software, incluindo manutenção
- Custos de viagens e treinamentos
- Custos de esforço (os custos de pagamento dos engenheiros de software).

Para a maioria dos projetos, o custo dominante é o custo de esforço. Os computadores suficientemente potentes para o desenvolvimento de software são relativamente baratos. Embora os custos de viagens extensivas sejam necessários, quando um projeto é desenvolvido em locais diferentes, os custos de viagem são geralmente uma pequena fração dos custos de esforço. Além disso, o uso de sistemas de comunicações eletrônicas, como e-mail, sites Web compartilhados e videoconferência, podem reduzir significativamente as viagens necessárias. A conferência eletrônica também significa que o tempo de viagem é reduzido e esse tempo pode ser usado com mais produtividade no desenvolvimento de software. Em um projeto no qual trabalhei, os custos de viagem e tempo foram reduzidos em quase 50% pelo fato de as reuniões serem realizadas por meio de videoconferência em vez de serem reuniões do tipo pessoal.

Os custos de esforço não são apenas salários dos engenheiros de software que estão envolvidos no projeto. As organizações calculam os custos de esforço em termos de custos indiretos nos quais é considerado o custo total de operação da organização e este é dividido pelo número de pessoal produtivo. Portanto, os seguintes custos são partes do custo total de esforço:

1. Custos de subsistência, aquecimento e iluminação no espaço de escritório
2. Custos de pessoal de apoio como contadores, administradores, gerentes de sistema, faxineiros e técnicos
3. Custos de operações de rede de comunicações
4. Custos de instalações centrais, como uma biblioteca ou de recreação
5. Custos de Seguridade Social e benefícios de empregado, como pensões e seguro-saúde.

Esse fator de custo indireto é, geralmente, pelo menos duas vezes o salário de engenheiro de software, dependendo do tamanho da organização e seus custos indiretos associados. Portanto, se uma empresa paga a um engenheiro de software 90 mil dólares por ano, os custos totais são pelo menos 180 mil dólares por ano ou 15 mil dólares por mês.

Uma vez que o projeto esteja em andamento, os gerentes de projeto devem atualizar regularmente suas estimativas de custos e de prazo. Isso ajuda no processo de planejamento e uso eficiente dos recursos. Se as despesas reais são significativamente maiores do que as estimadas, o gerente de projeto deve tomar alguma providência. Isso pode envolver a aplicação de recursos adicionais para o projeto ou modificação do trabalho a ser realizado.

O cálculo de custo de software deve ser realizado objetivamente com o propósito de prever precisamente o custo de desenvolvimento do software. Se o custo de projeto foi calculado como parte de um projeto oferecido a um cliente, uma decisão deve ser tomada sobre o preço cotado para ele. Em geral, o preço é, simplesmente, custo mais lucro. Contudo, o relacionamento entre o custo de projeto e o preço para o cliente não é, normalmente, tão simples.

A atribuição de preço de software deve levar em conta considerações mais amplas sobre questões organizacionais, econômicas, políticas e de negócios, como as mostradas na Tabela 26.1. Portanto, pode não haver um relacionamento simples entre o preço do software para o cliente e os custos de desenvolvimento de software. Por causa das considerações organizacionais envolvidas, a atribuição de preço de projeto deve envolver a gerência sênior (por exemplo, aqueles que tomam decisões estratégicas), bem como os gerentes de projeto de software.

Por exemplo, digamos que uma pequena empresa de software especializada em serviços de petróleo emprega 10 engenheiros no início de um ano, mas tem somente contratos em andamento que requerem 5 membros do pessoal de desenvolvimento. Contudo, ela está oferecendo para uma grande empresa de petróleo um contrato muito amplo que requer 30 pessoas-ano de esforço por 2 anos. O projeto não começará antes de pelo menos 12 meses, mas, se concedido, transformará as finanças da pequena empresa. A empresa de serviços de petróleo obtém uma oportunidade para oferecer um projeto que requer 6 pessoas e deve ser concluído em 10 meses. Os custos (entre eles custos indiretos desse projeto) são estimados em 1,2 milhões de dólares. Contudo, para melhorar a sua posição competitiva, a empresa de serviços de petróleo oferece um preço de 0,8 milhões de dólares para o cliente. Isso significa que, embora ele perca dinheiro neste contrato, pode reter o pessoal especialista para futuros projetos mais lucrativos.

26.1 Produtividade de software

Você pode medir a produtividade em um sistema de manufatura pela contagem do número de unidades produzidas e pela divisão do resultado da contagem pelo número de pessoas-hora necessárias para produzi-los. Contudo, para qualquer problema de software, existem muitas soluções diferentes, cada qual com atributos diferentes. Uma solução pode ser executada mais eficientemente enquanto outra pode ser mais legível e mais fácil para ser mantida. Quando soluções com atributos diferentes são produzidas, comparar suas taxas de produção não é realmente significativo.

Contudo, como um gerente de projeto, você pode estar diante do problema de estimativa de produtividade de engenheiros de software. Você pode necessitar dessas estimativas de produtividade para ajudar a definir o custo de projeto ou o prazo, para informar decisões de investimento ou para avaliar se as melhorias de processo ou de tecnologia foram eficazes.

As estimativas de produtividade são usualmente baseadas em medição de atributos de software e em divisão desses atributos pelo esforço total necessário para o seu desenvolvimento. Existem dois tipos de métricas que têm sido usados:

1. *Métricas relacionadas a tamanho*. Essas métricas estão relacionadas ao tamanho de algum resultado de uma atividade. Uma métrica relacionada a tamanho mais comumente usada é linhas de código-fonte entregues. Outra métrica que pode ser usada é o número de instruções de código objeto entregues ou número de páginas de documentação de sistema.
2. *Métricas relacionadas a funções*. Essas métricas são relacionadas à funcionalidade geral do software entregue. A produtividade é expressa em termos de quantidade de funcionalidade útil produzida em algum dado tempo. Pontos de função e pontos de objeto são as métricas mais bem conhecidas desse tipo.

Linhas de código-fonte por programador-mês (LOC/pm) é uma métrica de produtividade de software largamente usada. Você pode calcular LOC/pm pela contagem do número total de linhas de código-fonte a ser entregue e depois dividir o resultado pelo tempo total em programadores-mês para completar o projeto. Esse tempo, portanto, inclui o tempo necessário para todas as outras atividades (requisitos, projeto, codificação, testes e documentação) envolvidas no desenvolvimento de software.

Essa abordagem foi inicialmente desenvolvida quando a maioria das programações era feita em FORTRAN, linguagem Assembly ou COBOL. Nessa época, os programas eram perfurados em cartões, com uma declaração em cada cartão. O número de linhas de código era fácil de contar; ele correspondia ao número de cartões na caixa de programa. Contudo, os programas em linguagens, como Java e C++, consistem de declarações, comandos executáveis e comentários. Eles podem incluir instruções macro que se expandem para várias linhas de código. Pode haver mais do que uma declaração por linha. Não existe, portanto, um relacionamento simples entre as declarações de programa e as linhas de uma listagem.

A comparação de produtividade por meio de linguagens de programação pode dar impressões enganosas da produtividade do programador. Quanto mais expressiva for a linguagem de programação, menor será a produtividade aparente. Essa anomalia surge porque todas as atividades de desenvolvimento são consideradas quando se calcula o tempo de desenvolvimento, mas a métrica de LOC se aplica somente ao processo de programação. Portanto, se uma linguagem requer mais linhas de código do que outra para implementar a mesma funcionalidade, a estimativa de produtividade será anormal.

Por exemplo, considere um sistema em tempo real embutido que poderia ser codificado em 5.000 linhas de Assembly ou 1.500 linhas de C. O tempo de desenvolvimento, para as várias fases, é mostrado na Tabela 26.2. O programador de Assembly tem produtividade de 714 linhas/mês e um programador de linguagem de alto nível, menos do que a metade disso — 300 linhas/mês. Porém, os custos de desenvolvimento para sistemas desenvolvidos em C são menores e entregues antes.

Tabela 26.2 Tempos de desenvolvimento de sistema

	Análise	Projeto	Codificação	Testes	Documentação
Código Assembly	3 semanas	5 semanas	8 semanas	10 semanas	2 semanas
Linguagem de alto nível	3 semanas	5 semanas	4 semanas	6 semanas	2 semanas

	Tamanho	Esforço	Produtividade
Código Assembly	5.000 linhas	28 semanas	714 linhas/mês
Linguagem de alto nível	1.500 linhas	20 semanas	300 linhas/mês

Uma alternativa para o uso de tamanho de código como o atributo de produto estimado é usar alguma medida de funcionalidade do código. Isso evita as anomalias mostradas anteriormente, uma vez que a funcionalidade é independente da linguagem de implementação. MacDonell (MacDonell, 1994) descreve e compara brevemente várias medidas baseadas em funções. A mais conhecida dessas medidas é a contagem de pontos de função. Ela foi proposta por Albrecht (Albrecht, 1979) e refinada por Albrecht e Gaffney (Albrecht e Gaffney, 1983). Garmus e Herron (Garmus e Herron, 2000) descrevem o uso prático de pontos de função em projetos de software.

A produtividade é expressa como o número de pontos de função implementados por pessoa-mês. Um ponto de função não é uma característica única, porém é calculado por meio da combinação de várias medições ou estimativas. Você calcula o número total de pontos de função em um programa medindo ou estimando as seguintes características do programa:

- entradas e saídas externas;
- interações de usuários;
- interfaces externas;
- arquivos usados pelo sistema.

Obviamente, algumas entradas e saídas, interações e, assim por diante, são mais complexas que outras e levam mais tempo para implementar. A métrica de ponto de função leva isso em conta multiplicando a estimativa inicial de pontos de função por um fator ponderado de complexidade. Você deve avaliar cada uma dessas características de complexidade e então atribuir o fator ponderado que varia de 3 (para entradas externas simples) até 15 para arquivos internos complexos. Tanto os valores ponderados propostos por Albrecht como os valores baseados em experiência local podem ser usados.

Você pode então calcular o que se chamou de contagem de pontos de função não ajustada (UFC — *Unadjusted Function-point Count*), multiplicando cada contagem inicial pelo peso estimado e somando todos os valores.

UFC = Σ (número de elementos de dado tipo) x (peso)

Você então modifica essa contagem de ponto de função não ajustada por meio de fatores de complexidade adicionais que estão relacionados com complexidade do sistema como um todo. Isso leva em conta o grau de processamento distribuído, a quantidade de reuso, o desempenho, e assim por diante. A contagem de pontos de função não ajustada é multiplicada por esses fatores de complexidade de projeto para produzir a contagem de pontos de função final para o sistema geral.

Symons (Symons, 1988) observa que a natureza subjetiva da estimativa de complexidade significa que a contagem de pontos de função de um programa depende do estimador. Pessoas diferentes podem ter noções diferentes de complexidade. Existem, portanto, grandes variações na contagem de pontos de função dependendo do julgamento do estimador e do tipo do sistema a ser desenvolvido. Além disso, os pontos de função são voltados aos sistemas de processamento de dados dominados por operações de entrada e saída. É mais difícil estimar a contagem de pontos de função para sistemas orientados a eventos. Por essa razão, algumas pessoas pensam que a contagem de pontos de função não é uma maneira útil para medir a produtividade de software (Furey e Kitchenham, 1997; Armour, 2002). Contudo, os usuários de pontos de função argumentam que, apesar de suas imperfeições, eles são eficazes em situações práticas (Banker, et al., 1993; Germus e Herron, 2000).

Pontos de objeto (Banker, et al., 1994) são uma alternativa aos pontos de função. Eles podem ser usados com linguagens, como linguagens de programação de banco de dados ou linguagens de scripting. Pontos de objeto não são classes de objetos que podem ser produzidos quando uma abordagem orientada a objetos é adotada no desenvolvimento de software. Em vez disso, o número de pontos de objeto em um programa é uma estimativa ponderada de:

1. *Número de telas separadas exibidas*. Telas simples contam como 1 ponto de objeto, telas moderadamente complexas contam como 2 e telas muito complexas contam como 3 pontos de objeto.

2. *Número de relatórios produzidos*. Para relatórios simples, contar 2 pontos de objetos; para relatórios moderadamente complexos, contar 5 e, para relatórios provavelmente difíceis de serem produzidos, contar 8 pontos de objetos.
3. *Número de módulos em linguagens de programação imperativas, como Java ou C++, que deve ser desenvolvido para complementar o código de programação de banco de dados*. Cada um desses módulos conta com 10 pontos de objeto.

Os pontos de objeto são usados no modelo de estimativa COCOMO II (no qual são chamados pontos de aplicação) que abordo mais adiante neste capítulo. A vantagem dos pontos de objeto em relação aos pontos de função é que eles são mais fáceis de estimar com base em uma especificação de software em alto nível. Os pontos de objeto se dedicam somente a telas, relatórios e módulos em linguagens de programação convencionais. Não se dedicam a detalhes de implementação e a estimativa do fator de complexidade é muito mais simples.

Se pontos de função ou pontos de objeto forem usados, eles podem ser estimados nos estágios iniciais do processo de desenvolvimento antes que as decisões que afetem o tamanho de programa sejam tomadas. As estimativas desses parâmetros podem ser feitas assim que as interações externas do sistema forem projetadas. Nesse estágio, é muito difícil produzir uma estimativa precisa de tamanho de um programa em linhas de código-fonte.

As contagens de pontos de função e pontos de objeto podem ser usadas em conjunto com modelos de estimativas de linhas de códigos. O tamanho de código final é calculado baseado no número de pontos de função. Usando a análise de dados históricos, o número médio de linhas de código (AVC — *Average Number of Lines of Code*) necessário para implementar um ponto de função, em determinada linguagem, pode ser estimado. Os valores de AVC variam de 200 a 300 LOC/FP na linguagem Assembly para 2 a 4 LOC/FP em uma linguagem de banco de dados, como SQL. O tamanho de código estimado para uma nova aplicação é então calculado conforme segue:

Tamanho de código = AVC × Número de pontos de função

A produtividade de programação de indivíduos que trabalham em uma organização é afetada por uma série de fatores. Alguns dos fatores mais importantes estão resumidos na Tabela 26.3. Contudo, diferenças individuais, em termos de capacidade, são mais significativas do que qualquer um desses fatores. Em uma avaliação preliminar de produtividade, Sackman et al. (Sackman, et al., 1968) descobriram que alguns programadores eram, pelo menos, 10 vezes mais produtivos do que outros. Minha experiência diz que isso ainda é verdade. As grandes equipes, provavelmente, apresentam uma mistura de capacidades e experiências e, portanto, terão uma produtividade 'média'. Em pequenas equipes, contudo, a produtividade geral é bastante dependente das aptidões e capacidades individuais.

A produtividade de desenvolvimento de software varia dramaticamente em domínios de aplicação e organizações. Para sistemas grandes, complexos, embarcados, a produtividade estimada tem sido tão pequena quanto 30 LOC/pm. Para sistemas de aplicação simples, bem definida, escritos em uma linguagem, como Java, a produtividade pode ser tão alta quanto 900 LOC/pm. Quando medidos em termos de pontos de objeto, Boehm et al. (Boehm, et al., 1995) sugerem que a produtividade varie de 4 pontos de objeto por mês até 50 pontos de objetos por mês, dependendo do tipo de aplicação, apoio de ferramentas e capacidade do desenvolvedor.

O problema com medidas que contam com a quantidade produzida em um dado período de tempo é que eles não levam em conta as características de qualidade, como confiabilidade e facilidade de manutenção. Eles implicam que mais é sempre melhor.

Tabela 26.3 Fatores que afetam a produtividade de engenharia de software

Fator	Descrição
Experiência no domínio da aplicação	O conhecimento do domínio da aplicação é essencial para o desenvolvimento eficiente de software. Os engenheiros que já compreendem um domínio serão, provavelmente, os mais produtivos.
Qualidade de processo	O processo de desenvolvimento usado pode ter efeito significativo sobre a produtividade. Esse tópico é abordado no Capítulo 28.
Tamanho de projeto	Quanto maior o projeto, mais tempo é necessário para a comunicação da equipe. Se menos tempo estiver disponível para o desenvolvimento, a produtividade individual será reduzida.
Apoio de tecnologia	Uma boa tecnologia de apoio, como ferramentas CASE e sistemas de gerenciamento de configurações, pode aumentar a produtividade.
Ambiente de trabalho	Conforme explicado no Capítulo 25, um ambiente de trabalho calmo com áreas de trabalho privativas contribui para o aumento da produtividade.

Beck (Beck, 2000), em sua explicação sobre *extreme programming*, faz uma excelente observação sobre estimativa. Se sua abordagem é baseada em simplificação e melhoria contínua de código, a contagem de linhas de código não significa muito.

Estas medidas também não levam em conta a possibilidade de reuso de software produzido, de uso de geradores de código e outras ferramentas que ajudam a criar o software. O que realmente queremos estimar é o custo de derivar um sistema específico com dada funcionalidade, qualidade, desempenho, facilidade de manutenção etc. Isso está somente relacionado indiretamente com medidas tangíveis, como tamanho de sistema.

Como gerente, você não deve usar as medições de produtividade para fazer julgamentos apressados sobre as capacidades dos engenheiros da sua equipe. Se você fizer isso, os engenheiros podem comprometer a qualidade, a fim de tornarem-se mais produtivos. Esse pode ser o caso quando um programador 'menos produtivo' produz código mais confiável — código mais fácil de compreender e mais barato de manter. Você deve sempre, portanto, pensar em medidas de produtividade como informações parciais sobre a produtividade do programador. Você também precisa considerar outras informações sobre a qualidade dos programas produzidos.

26.2 Técnicas de estimativa

Não existe uma maneira simples de fazer uma estimativa precisa do esforço necessário para desenvolver um sistema de software. Você talvez tenha de fazer uma estimativa inicial com base em uma definição de requisitos de usuário de mais alto nível. É possível que o software tenha de operar em computadores não familiares ou usar uma nova tecnologia de desenvolvimento. As pessoas envolvidas no projeto e suas habilidades, não serão conhecidas provavelmente. Tudo isso significa que é impossível estimar os custos de desenvolvimento de sistema precisamente nas fases preliminares de um projeto.

Além disso, existe uma dificuldade fundamental em avaliar a precisão das diferentes abordagens para as técnicas de estimativa de custos. Estimativas de custo de projeto são, freqüentemente, auto-suficientes. A estimativa é usada para definir o orçamento do projeto e o produto é ajustado de modo que o orçamento seja cumprido. Não conheço nenhum experimento controlado com custos de projeto no qual os custos estimados não foram usados para influenciar o experimento. Um experimento controlado não revelaria o custo estimado para o gerente de projeto. Os custos reais então poderiam ser comparados com os custos estimados de projeto. Contudo, tal experimento é provavelmente impossível devido aos altos custos envolvidos e ao número de variáveis que não podem ser controladas.

Por outro lado, as organizações necessitam estimar esforço e custo de software. Para fazer isso, uma ou mais das técnicas descritas na Tabela 26.4 podem ser usadas (Boehm, 1981). Todas essas técnicas contam com os julgamentos dos gerentes de projeto centrados na experiência que usam seus conhecimentos de projetos anteriores para chegar a uma estimativa de recursos necessários para o projeto. Contudo, pode haver diferenças importantes entre os projetos passados e futuros. Muitos métodos e técnicas novas de desenvolvimento foram introduzidos nos últimos 10 anos. Alguns exemplos de mudanças que podem afetar as estimativas baseadas em experiências incluem:

1. Sistemas de objetos distribuídos em vez de sistemas baseados em mainframes
2. Uso de Web services

Tabela 26.4 Técnicas de estimativas de custos

Técnica	Descrição
Modelagem algorítmica de custos	Um modelo é desenvolvido usando informações históricas de custos que relacionam alguma métrica de software (geralmente, seu tamanho) aos custos de projeto. É feita uma estimativa dessa métrica e o modelo prevê o esforço necessário.
Julgamento de especialista	Vários especialistas nas técnicas de desenvolvimento de software propostas e no domínio da aplicação são consultados. Cada um deles estima o custo do projeto. Essas estimativas são comparadas e discutidas. O processo de estimativa prossegue iterativamente até que seja atingida uma estimativa em comum.
Estimativa por analogia	Essa técnica é aplicável quando outros projetos do mesmo domínio de aplicação foram concluídos. O custo de um novo projeto é estimado por analogia com esses projetos concluídos. Myers (Myers, 1989) fornece uma clara descrição dessa abordagem.
Lei de Parkinson	A Lei de Parkinson estabelece que o trabalho se expande para preencher o tempo disponível. O custo é determinado pelos recursos disponíveis e não pela avaliação objetiva. Se o software deve ser entregue em 12 meses e 5 pessoas estão disponíveis, o esforço necessário é estimado em 60 pessoas-mês.
Atribuição de preço para ganhar	O custo de software é estimado para ser o que o cliente tiver disponível para gastar no projeto. O esforço estimado depende do orçamento do cliente, e não da funcionalidade do software.

3. Uso de ERP ou sistemas centrados em banco de dados
4. Uso de software comercial em vez do desenvolvimento de um novo sistema
5. Desenvolvimento para e com reuso em vez de novo desenvolvimento de todas as partes de um sistema
6. Desenvolvimento com uso de linguagens de scripting, como o TCL ou Perl (Ousterhout, 1998)
7. Uso de ferramentas CASE e geradores de programa em vez de desenvolvimento de software sem apoio.

Se os gerentes de projeto não trabalharam com essas técnicas, suas experiências anteriores podem não ajudá-los a estimar os custos de projeto de software. Isso torna mais difícil para eles produzirem estimativas precisas de custos e de prazo.

Você pode lidar com as abordagens para estimativas de custo conforme mostradas na Tabela 26.4 usando uma abordagem top-down ou bottom-up. Uma abordagem top-down começa no nível de sistema. Você começa examinando a funcionalidade geral do produto e como essa funcionalidade é fornecida por meio de interações entre subfunções. Os custos das atividades no nível de sistema, como integração, gerenciamento de configuração e documentação são levados em conta.

A abordagem bottom-up, por outro lado, começa no nível de componente. O sistema é decomposto em componentes e você estima o esforço necessário para desenvolver cada um desses componentes. Você, então, adiciona os custos de componentes para calcular o esforço necessário para o desenvolvimento de todo o sistema.

As desvantagens da abordagem top-down são as vantagens da abordagem bottom-up e vice-versa. A estimativa top-down pode subestimar os custos de resolução de problemas técnicos difíceis, associados com componentes específicos, como interfaces para hardware não padronizado. Não existe uma justificativa detalhada de estimativa produzida. Por outro lado, a estimativa bottom-up produz uma justificativa que considera cada componente. Contudo, essa abordagem, provavelmente, poderá subestimar os custos de atividades de sistema, como a de integração. A estimativa bottom-up é também mais onerosa. Deve existir um projeto inicial de sistema para identificar os componentes a serem custeados.

Cada técnica de estimativa tem os seus próprios pontos fortes e fracos. Cada uma usa informações diferentes sobre o projeto e sobre a equipe de desenvolvimento, de modo que, se você usa um único modelo e essas informações não são precisas, sua estimativa final poderá estar errada. Para projetos de grande porte, portanto, você deve usar várias técnicas de estimativas e comparar os seus resultados. Se essas técnicas previrem custos radicalmente diferentes, você, provavelmente, não teve informações suficientes sobre o produto ou sobre o processo de desenvolvimento. Você deve procurar mais informações sobre o produto, processo ou equipe e repetir o processo até que as estimativas sejam convergentes.

Essas técnicas de estimativas são aplicáveis quando um documento de requisitos para sistema foi produzido. Esse documento deve definir todos os requisitos de usuário e de sistema. Você pode, portanto, fazer uma estimativa razoável da funcionalidade de sistema que será desenvolvida. Em geral, grandes projetos de engenharia de sistemas terão um documento de requisitos.

Contudo, em muitos casos, os custos de muitos projetos devem ser estimados usando somente os requisitos incompletos de usuário para o sistema. Isso significa que os estimadores têm muito poucas informações com que trabalhar. Análise e especificação de requisitos são dispendiosas e os gerentes de uma empresa podem precisar uma estimativa de custo inicial para o sistema antes que eles possam ter um orçamento aprovado para desenvolver requisitos mais detalhados ou um protótipo de sistema.

Sob essas circunstâncias, a atribuição de 'preço para ganhar' é uma estratégia comumente usada. A noção de atribuição 'preço para ganhar' pode parecer antiética e inadequada para negócios. Contudo, isso tem algumas vantagens. Um custo de projeto é acordado com base em um esboço de proposta. As negociações portanto ocorrem entre o cliente e o fornecedor para estabelecer a especificação detalhada de projeto. Essa especificação é restrita pelo custo acordado. O comprador e o vendedor devem concordar sobre o que é aceitável na funcionalidade de sistema. O fator fixado em muitos projetos não são os requisitos de projeto, mas sim o custo. Os requisitos podem ser alterados, de modo que o custo não pode ser excedido.

Por exemplo, digamos que uma empresa está oferecendo um contrato de desenvolvimento de um novo sistema de entrega de combustível para uma empresa de petróleo que programa as entregas de combustível para os seus postos de gasolina. Não existe um documento detalhado de requisitos para esse sistema e, assim, os desenvolvedores estimam que um preço de 900 mil dólares é, provavelmente, competitivo e fica dentro do orçamento da empresa de petróleo. Depois que eles concederam o contrato, negociaram os requisitos detalhados do sistema de modo que a funcionalidade básica fosse entregue; depois eles estimaram os custos adicionais para outros requisitos. A empresa de petróleo não perdeu necessariamente neste contrato porque ela foi premiada com um contrato para uma empresa em que ele podia confiar. Os requisitos adicionais podem ser considerados em um orçamento futuro, de modo que a elaboração de orçamento da empresa de petróleo não é interrompida por um custo inicial de software muito alto.

26.3 Modelagem algorítmica de custos

A modelagem algorítmica de custos usa uma fórmula matemática para prever custos baseados em estimativas sobre o tamanho de projeto, o número de engenheiros de software e outros fatores de processo e de produto. Um modelo

algorítmico de custos pode ser construído analisando-se os custos e os atributos de projetos completados e encontrando-se a fórmula adequada mais próxima da experiência real.

Os modelos algorítmicos de custos são, normalmente, usados para fazer estimativas de custos de desenvolvimento de software, mas Boehm (Boehm, et al., 2000) explica vários outros usos para as estimativas algorítmicas de custos, entre eles as estimativas de investidores em empresas de software, as estimativas de estratégias alternativas para ajudar a avaliar riscos e estimativas para informar decisões sobre reuso, novo desenvolvimento ou terceirização.

Na sua forma mais geral, uma estimativa algorítmica de custos, para custo de software, pode ser expressa como:

$$\textbf{Esforço} = \textbf{A} \times \textbf{Tamanho}^{\textbf{B}} \times \textbf{M}$$

A é um fator constante que depende das práticas organizacionais locais e o tipo de software desenvolvido. **Tamanho** pode ser tanto uma avaliação de tamanho de código como uma estimativa de funcionalidade, expressa em pontos de função ou de objeto. O valor do expoente **B** usualmente fica entre 1 e 1,5. **M** é um multiplicador constituído pela combinação de processo, produto e atributos de desenvolvimento, como requisitos de confiança para o software e a experiência da equipe de desenvolvimento.

A maioria dos modelos algorítmicos de estimativas tem um componente exponencial (**B** na equação anterior) associado à estimativa de tamanho. Isso reflete o fato de que os custos, normalmente, não aumentam de maneira linear com o tamanho de projeto. À medida que o software cresce, custos extras são incorridos por causa do custo indireto de comunicação de equipes grandes, gerenciamento de configuração mais complexo, integração de sistema mais difícil etc. Portanto, quanto maior é o sistema, maior é o valor desse expoente.

Infelizmente, todos os modelos algorítmicos sofrem das mesmas dificuldades fundamentais:

1. *É difícil muitas vezes estimar* **Tamanho** *no estágio inicial de um projeto quando somente uma especificação está disponível.* As estimativas de pontos de função e pontos de objeto são mais fáceis de produzir do que as estimativas de tamanho de código, mas elas são ainda, freqüentemente, imprecisas.
2. *As estimativas dos fatores que contribuem para* **B** *e* **M** *são subjetivas* As estimativas variam de uma pessoa para outra, dependendo da sua formação e experiência com o tipo de sistema em desenvolvimento.

O número de linhas de código-fonte no sistema entregue é a métrica básica usada em muitos modelos algorítmicos de custos. A estimativa de tamanho pode envolver a estimativa por analogia com outros projetos, a estimativa por conversão de pontos de função ou pontos de objetos para tamanho de código, a estimativa por classificação de tamanhos de componentes de sistema e por uso de um componente de referência conhecida, para estimar o tamanho do componente, ou pode ser simplesmente uma questão de julgamento de engenharia.

A estimativa precisa de tamanho de código é difícil em um estágio inicial de projeto, pois o tamanho de código é afetado por decisões de projeto que ainda não foram tomadas. Por exemplo, uma aplicação que requer um gerenciamento de dados complexo pode usar um banco de dados comercial ou implementar o seu próprio sistema de gerenciamento de dados. Se um banco de dados comercial for usado, o tamanho de código será menor, mas um esforço adicional pode ser necessário para superar as limitações de desempenho do produto comercial.

A linguagem de programação usada para o desenvolvimento do sistema também afeta o número de linhas de código a ser desenvolvido. Uma linguagem, como Java, poderia significar que mais linhas de código são necessárias do que se C (digamos) fosse usada. Contudo, esse código extra permite mais verificação em tempo de compilação e, assim, os custos de validação, provavelmente, serão reduzidos. Como isso deve ser levado em consideração? Além do mais, pode ser possível reusar uma quantidade significativa de código de projetos anteriores e a estimativa de tamanho deve ser ajustada para levar isso em consideração.

Se você usa um modelo algorítmico de estimativa de custos, deve desenvolver várias estimativas (a pior, a esperada e a melhor) em vez de uma única estimativa e aplicar a fórmula de custos em todas elas. As estimativas podem ser precisas se você compreende o tipo de software em desenvolvimento quando ajustou o modelo de custo usando os dados locais e quando as opções de linguagem de programação e de hardware foram predefinidas.

A precisão de estimativas produzidas por um modelo algorítmico depende do sistema de informações disponível. À medida que o processo de software progride, mais informações se tornam disponíveis e, assim, as estimativas se tornam cada vez mais precisas. Se a estimativa inicial de esforço necessário é de x meses de esforço, a variação pode ser de $0{,}25x$ a $4x$, quando o sistema é proposto pela primeira vez. Essa variação se estreita durante o processo de desenvolvimento, conforme mostrado na Figura 26.1. Essa figura, adaptada do artigo de Boehm (Boehm, et al., 1995), reflete a experiência de um grande número de projetos de desenvolvimento de software. Naturalmente, um pouco antes da entrega do sistema, uma estimativa muito precisa pode ser feita.

26.3.1 O modelo COCOMO

Uma série de modelos algorítmicos foi proposta como base para estimativa de esforço, prazos e custos de um projeto de software. Esses modelos são conceitualmente similares, mas usam valores diferentes de parâmetros. O modelo

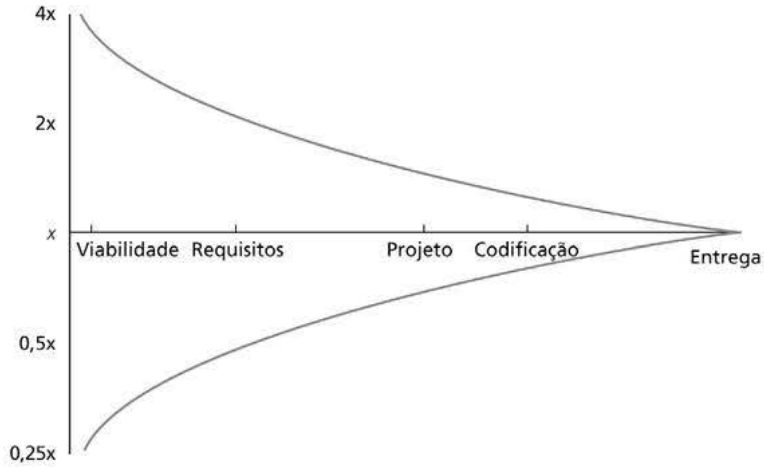


Figura 26.1
Incerteza de estimativa.

que explico nesta seção é o modelo COCOMO (*Construtive Cost Model*). O modelo COCOMO é um modelo empírico que foi criado por meio de coleta de dados com base em um grande número de projetos de software. Esses dados foram analisados para descobrir fórmulas que fossem as mais adequadas às observações. Tais fórmulas ligam o tamanho do sistema e do produto, fatores de projeto e de equipe ao esforço para desenvolver o sistema.

Escolhi usar o modelo COCOMO por várias razões:

1. Ele é bem documentado, disponível, de domínio público e apoiado por ferramentas comerciais e de domínio público.
2. Ele tem sido amplamente usado e avaliado em várias organizações.
3. Ele tem uma longa linhagem, desde a sua primeira versão, lançada em 1981 (Boehm, 1981), seguindo um refinamento ajustado ao desenvolvimento de software em ADA (Boehm e Royce, 1989) até a sua mais recente versão, o COCOMO II, publicado em 2000 (Boehm, et al., 2000).

Os modelos COCOMO são abrangentes e têm grande número de parâmetros, sendo que cada um pode ganhar uma variedade de valores. Eles são tão complexos que não posso dar uma descrição completa nesta seção. Em vez disso, explico simplesmente suas características essenciais para dar um entendimento básico dos modelos algorítmicos de custos.

A primeira versão do modelo COCOMO (COCOMO 81) era um modelo de três níveis, sendo que os níveis correspondiam aos detalhes da análise de estimativa de custos. O primeiro nível (básico) fornecia uma estimativa inicial, preliminar; o segundo nível modificava isso usando uma série de multiplicadores de projeto e de processo; e o nível mais detalhado produzia estimativas para as diferentes fases do projeto. A Tabela 26.5 mostra a fórmula básica do COCOMO para diferentes tipos de projetos. O multiplicador **M** reflete as características de produto, de projeto e de equipe.

O COCOMO 81 considerava que o software seria desenvolvido de acordo com um processo cascata (veja o Capítulo 4) usando linguagens de programação imperativa padronizadas, como C ou FORTRAN. Contudo, tem havido mudanças radicais no desenvolvimento de software desde que a sua versão inicial foi proposta. A prototipação e o desenvolvimento incremental são modelos de processo comumente usados. Atualmente, o software é desenvolvido, muitas vezes, por meio da montagem de componentes reusáveis e sistemas comerciais, e 'colando'-os com linguagem de scripting. Sistemas de uso intensivo de dados são desenvolvidos com a utilização de linguagens de programação de banco de dados, como o SQL, e um sistema gerenciador de banco de dados comercial. O software existente sofre reengenharia para criar um novo software. As ferramentas CASE que apóiam a maioria das atividades de processo de software está atualmente disponível.

Tabela 26.5 Modelo COCOMO 81 básico

Complexidade do projeto	Fórmula	Descrição
Simples	$PM = 2{,}4\ (KDSI)^{1{,}05} \times M$	Aplicações bem compreendidas, desenvolvidas por pequenas equipes.
Moderada	$PM = 3{,}0\ (KDSI)^{1{,}12} \times M$	Projetos mais complexos em que os membros da equipe podem ter experiência limitada com os sistemas relacionados.
Incorporada	$PM = 3{,}6\ (KDSI)^{1{,}20} \times M$	Projetos complexos nos quais o software é parte de um conjunto complexo e fortemente acoplado de hardware, software, leis e procedimentos operacionais.

Para levar essas mudanças em consideração, o modelo COCOMO II reconhece abordagens diferentes para o desenvolvimento de software, como prototipação, desenvolvimento por composição de componentes e uso de programação de banco de dados. O COCOMO II apóia o modelo espiral de desenvolvimento (veja o Capítulo 4) e incorpora vários submodelos que produzem, incrementalmente, estimativas detalhadas. Esses modelos podem ser usados em rodadas sucessivas do desenvolvimento espiral. A Figura 26.2 mostra os submodelos do COCOMO II e quando eles são usados.

Os submodelos que são parte do modelo COCOMO II são:

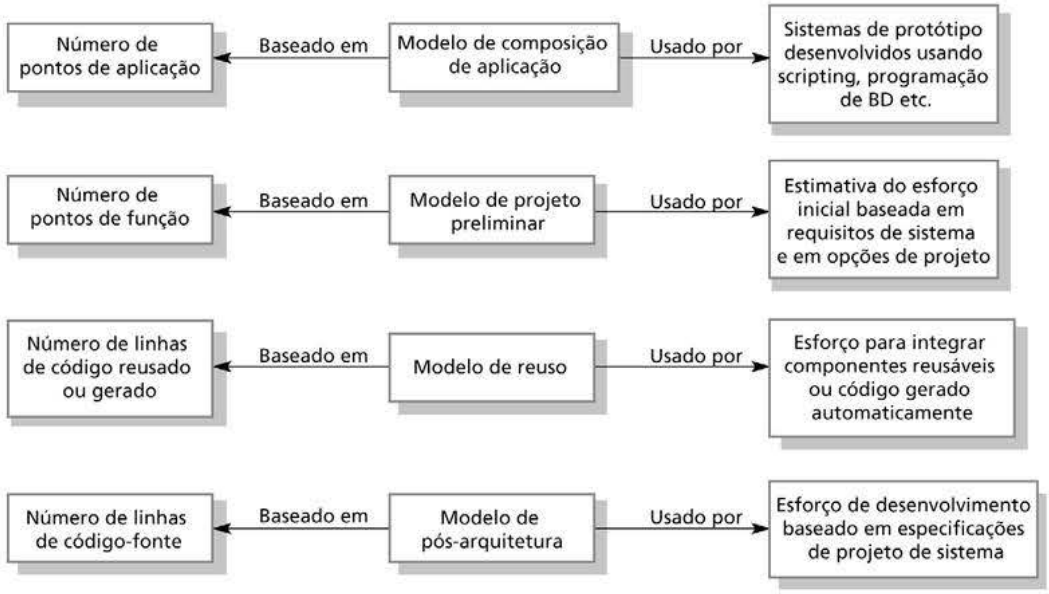
1. *Um modelo de composição de aplicação*. Esse modelo considera que os sistemas são criados baseados em componentes reusáveis, programação por scripting ou de banco de dados. Ele é projetado para fazer estimativas de desenvolvimento de protótipo. As estimativas de tamanho de software são baseadas em pontos de aplicação, nos quais uma fórmula simples de tamanho/produtividade é usada para estimar o esforço necessário. Os pontos de aplicação são os mesmos pontos de objeto explicados na Seção 26.1, mas o nome foi mudado para evitar confusão com objetos no desenvolvimento orientado a objetos.
2. *Um modelo de projeto preliminar*. Esse modelo é usado durante os estágios iniciais do projeto de sistema depois que os requisitos foram estabelecidos. As estimativas são baseadas em pontos de função, que são, depois, convertidas em número de linhas de código-fonte. A fórmula segue a forma-padrão, conforme explicado anteriormente, com um conjunto simplificado de sete multiplicadores.
3. *Um modelo de reuso*. Esse modelo é usado para calcular o esforço necessário e para integrar componentes reusáveis e/ou código de programa gerado automaticamente por meio de ferramentas de projeto ou de tradução de programas. É geralmente usado em conjunto com o modelo pós-arquitetura.
4. *Um modelo de pós-arquitetura*. Uma vez que a arquitetura de sistema foi projetada, uma estimativa mais precisa de tamanho de software pode ser feita. Novamente, esse modelo usa a fórmula-padrão para estimativa de custos, explicada anteriormente. Entretanto, ele inclui um conjunto mais extenso de 17 multiplicadores que refletem a capacitação pessoal e as características de produto e de projeto.

Naturalmente, em sistemas de grande porte, partes diferentes podem ser desenvolvidas com o uso de tecnologias diferentes e, possivelmente, você não precisa estimar todas as partes do sistema no mesmo nível de precisão. Nesses casos você pode usar o submodelo apropriado para cada parte do sistema e combinar os resultados para criar uma estimativa composta.

Modelo de composição de aplicação

O modelo de composição de aplicação foi introduzido no COCOMO II para apoiar a estimativa de esforço necessário para projetos de prototipação e para projetos em que o software é desenvolvido pela composição de componentes existentes. Ele baseia-se na estimativa de pontos de aplicação (pontos de objetos) ponderada dividido por uma estimativa padrão de produtividade em pontos de aplicação. A estimativa é, depois, ajustada de acordo com a dificuldade de desenvolver cada ponto de objeto (Boehm, et al., 2000). A produtividade de programador também depende da experiência e capacitação do

Figura 26.2 Modelos COCOMO II.



desenvolvedor bem como da capacidade das ferramentas CASE usadas para apoiar o desenvolvimento. A Tabela 26.6 mostra o nível da produtividade em pontos de objeto (NOP/mês – Número de Pontos de Objeto/mês) sugerido pelos desenvolvedores do modelo (Boehm, et al., 1995).

A composição de aplicação envolve, normalmente, reuso significativo de software e alguns desses números totais de pontos de aplicação no sistema podem ser implementados com componentes reusáveis. Conseqüentemente, você precisa ajustar a estimativa baseada no número total de pontos de aplicação para levar em consideração o percentual de reuso esperado. Portanto, a fórmula final para o cálculo de esforço para protótipos de sistema é:

PM = (NAP × (1 – %reuso/100))/PROD

PM é o esforço estimado em pessoa-mês. **NAP** é o número total de pontos de aplicação no sistema entregue. **%reuso** é uma estimativa de quantidade de código reusado no desenvolvimento. **PROD** é a produtividade em pontos de objeto, conforme mostrada na Tabela 26.6. O modelo supõe de maneira simplista que não existe esforço adicional envolvido no reuso.

Modelo de projeto preliminar

Este modelo é usado quando os requisitos de usuário foram acordados e os estágios iniciais do processo de projeto foram realizados. Contudo, você não necessita de um projeto de arquitetura detalhado para fazer essas estimativas iniciais. Sua meta, nesse estágio, deve ser a de fazer uma estimativa aproximada sem esforço excessivo. Conseqüentemente, você faz várias suposições simplificadoras, como a do esforço envolvido na integração de código reusável ser zero. As estimativas de o de projeto preliminares são mais úteis como exploração de opções, nas quais você precisa comparar diferentes modos de implementação de requisitos de usuário.

As estimativas produzidas nesse estágio baseiam-se na fórmula-padrão para modelos algorítmicos, que é:

Esforço = A x Tamanho^B x M

Baseado no seu próprio conjunto grande de dados, Boehm propôs que o coeficiente de **A** deve ser 2,94. O tamanho do sistema é expresso em KSLOC, que é o número em milhares de linhas de código-fonte. Você calcula o KSLOC estimando o número de pontos de função no software. Usa, então, tabelas-padrão que relacionam o tamanho de software a pontos de função para diferentes linguagens de programação, para calcular uma estimativa inicial do tamanho de sistema em KSLOC.

O expoente **B** reflete o crescimento de esforço necessário à medida que cresce o tamanho do projeto. Esse expoente não é fixado para tipos diferentes de sistemas, como em COCOMO 81, mas pode variar de 1,1 a 1,24, dependendo do grau de inovação do projeto, flexibilidade de desenvolvimento, processos de resolução de risco usados, a coesão da equipe de desenvolvimento e o nível de maturidade de processo (veja o Capítulo 28) da organização. Explico como o valor desse expoente é calculado usando esses parâmetros na descrição do modelo de pós-arquitetura do COCOMO II.

O multiplicador **M**, no COCOMO II, baseia-se em um conjunto simplificado de sete características de projeto e de processo que influenciam a estimativa que podem aumentar ou diminuir o esforço necessário. Essas características usadas no modelo de projeto preliminar são a confiabilidade e a complexidade de produto (**RCPX**), o reuso necessário (**RUSE**), a dificuldade de plataforma (**PDIF**), a capacidade de pessoal (**PERS**), a experiência do pessoal (**PREX**), o prazo (**SCED**) e os recursos de apoio (**FCIL**). Você estima os valores desses atributos usando uma escala de seis pontos, sendo que 1 corresponde a valores muito baixos para esses multiplicadores e 6 corresponde a valores muito altos.

Isso resulta em um cálculo de esforço, conforme segue:

PM = 2,94 × Tamanho^B x M

sendo:

M = PERS × RCPX × RUSE × PDIF × PREX × FCIL × SCED

Modelo de reuso

Conforme expliquei nos capítulos 18 e 19, o reuso de software é comum atualmente e a maioria dos sistemas de grande porte inclui um percentual significativo de código reusado com base em desenvolvimentos anteriores. O modelo de reuso é usado para estimar o esforço necessário para integrar código gerado e reusável.

Tabela 26.6 Produtividade em pontos de objeto

Experiência e capacitação do desenvolvedor	Muito baixa	Baixa	Nominal	Alta	Muito alta
Maturidade e capacitação de CASE	Muito baixa	Baixa	Nominal	Alta	Muito alta
PROD (NOP/mês)	4	7	13	25	50

O COCOMO II considera dois tipos de código reusado. Código caixa-preta é o código que pode ser reusado sem compreendê-lo ou fazer mudanças nele. O esforço de desenvolvimento para códigos caixa-preta é considerado zero. O código que deve ser adaptado para integrá-lo com códigos novos ou outros componentes reusados é chamado de código caixa-branca. Algum esforço de desenvolvimento é necessário para reusar esse código, porque ele precisa ser compreendido e modificado antes que possa funcionar corretamente no sistema.

Adicionalmente, muitos sistemas incluem automaticamente código gerado de tradutores de programas, que geram códigos baseados em modelos de sistemas. Essa é uma maneira de reuso em que os templates-padrão estão embutidos no gerador. O modelo de sistema é analisado e o código é gerado, com base nesses templates-padrão com detalhes adicionais do modelo de sistema. O modelo de reuso do COCOMO II inclui um modelo separado para estimar os custos associados com esse código gerado.

Para o código gerado automaticamente, o modelo estima o número de pessoas-mês necessário para integrá-lo. A fórmula para estimativa de esforço é:

$$PM_{Auto} = (ASLOC \times AT/100)/ATPROD \qquad \text{// Estimativa para o código gerado}$$

AT é um percentual de código adaptado gerado automaticamente, e **ATPROD** é a produtividade dos engenheiros na integração de tal código. Boehm et al. (Boehm, et al., 2000) mediram **ATPROD** para ser em torno de 2.400 declarações-fonte por mês. Portanto, se existe um total de 20.000 linhas de código caixa-branca reusado em um sistema, e 30% deste total é gerado automaticamente, o esforço necessário para integrar esse código gerado é:

$$(20.000 \times 30/100)/2.400 = 2{,}5 \text{ pessoa-mês} \qquad \text{// Código-exemplo gerado}$$

O outro componente do modelo de reuso é usado quando um sistema inclui algum código novo e alguns componentes caixa-branca que devem ser integrados. Nesse caso, o modelo de reuso não calcula o esforço diretamente. Em vez disso, baseado no número de linhas de código reusadas, ele calcula um valor que representa o número equivalente de linhas do código novo.

Portanto, se 30.000 linhas de código serão reusadas, a estimativa do novo tamanho equivalente poderia ser 6.000. Essencialmente, o reuso de 30.000 linhas de código é considerado equivalente a escrever 6.000 linhas de código novo. Esse valor calculado é adicionado ao número de linhas de código novo a ser desenvolvido no modelo de pós-arquitetura do COCOMO II.

As estimativas desse modelo de reuso são:

ASLOC — número de linhas de código nos componentes que devem ser adaptados;

ESLOC — número equivalente de linhas de código-fonte novo.

A fórmula usada para calcular **ESLOC** leva em consideração o esforço necessário para a compreensão de software, para implementação de mudanças no código reusado e no sistema para integrar esse código. Ele também leva em consideração a quantidade de código gerada automaticamente, na qual o esforço de desenvolvimento é calculado, conforme explicado anteriormente nesta seção.

A fórmula a seguir é usada para calcular o número de linhas equivalentes de código-fonte:

$$ESLOC = ASLOC \times (1 - AT/100) \times AAM$$

ASLOC é reduzido de acordo com o percentual de código gerado automaticamente. **AAM** é o Multiplicador de Ajuste de Adaptação, que leva em conta o esforço necessário para reusar o código. De maneira simplista, **AAM** é a soma de três componentes:

1. Um componente de adaptação (referido como **AAF**), que representa os custos de implementar mudanças para um código reusado. Isso inclui componentes que levam em consideração mudanças de projeto, de codificação e de integração.
2. Um entendimento de componente (referido como **SU**) que representa os custos de entendimento do código a ser reusado e a familiaridade do engenheiro com o código. **SU** varia de 50 para código complexo não estruturado a 10 para código bem escrito orientado a objetos.
3. Um fator de avaliação (referido como **AA**) que representa os custos de tomada de decisão de reuso. Ou seja, alguma análise é sempre necessária para decidir se o código pode ser reusado, e isso é incluído no custo como **AA**. **AA** varia de 0 a 8 dependendo da quantidade de esforço de análise necessária.

O modelo de reuso é um modelo não linear. Se o reuso é considerado, algum esforço é necessário para fazer uma avaliação, se o reuso for possível. Além disso, quanto mais reuso for contemplado, os custos por unidade de código reusado diminuem à medida que os custos de entendimento e avaliação são espalhados por meio de mais linhas de código.

Modelo de pós-arquitetura

O modelo de pós-arquitetura é o modelo mais detalhado dos modelos COCOMO II. É usado uma vez que esteja disponível um projeto de arquitetura inicial para o sistema de modo que se conheça a estrutura de subsistemas.

As estimativas produzidas no nível de pós-arquitetura são baseadas na mesma na fórmula básica (**PM = A x Tamanho^B x M**) usada nas estimativas de projeto anteriores. Contudo, a estimativa de tamanho para o software deve ser mais precisa nesse estágio do processo de estimativa. Além disso, um conjunto muito mais extenso de atributos de produto, de processo e organizacionais (17 em vez de 7) é usado para refinar o cálculo inicial de esforço. É possível usar mais atributos neste estágio porque você tem mais informações sobre o software a ser desenvolvido e o processo de desenvolvimento.

A estimativa de tamanho de código no modelo de pós-arquitetura é calculada usando três componentes:

1. Estimativa do número total de linhas do código novo a serem desenvolvidas
2. Estimativa do número equivalente de linhas de código-fonte (ESLOC) calculada usando o modelo de reuso
3. Estimativa do número de linhas de código que devem ser modificadas por causa das mudanças nos requisitos

Essas três estimativas são somadas para dar o total de tamanho de código em KSLOC, que você usa na fórmula de cálculo de esforço. O componente final na estimativa — o número de linhas de código modificadas — reflete o fato de que os requisitos de software sempre mudam. Os programas de sistema devem refletir essas mudanças de requisitos de modo que um código adicional deva ser desenvolvido. Naturalmente, a estimativa de número de linhas de códigos que serão alteradas não é fácil, e existirão mesmo, freqüentemente, mais incertezas a respeito desse valor do que das estimativas de desenvolvimento.

Havia três possibilidades de valores no COCOMO 1 para o termo expoente (**B**) na fórmula de cálculo de esforço. Esses valores eram relacionados aos níveis de complexidade de projeto. À medida que os projetos se tornaram mais complexos, os efeitos do aumento de tamanho de sistema se tornaram mais significativos. Contudo, boas práticas e procedimentos organizacionais podem controlar essa 'deseconomia' de escala. Isso é reconhecido no COCOMO II, sendo que a faixa de valores para o expoente **B** é contínua em vez de discreta. O expoente baseia-se em cinco fatores de escala, conforme mostrado na Tabela 26.7. Esses fatores são avaliados numa escala de seis pontos variando de Muito baixo para Extra alto (5 até 0). Você deve portanto adicionar as avaliações, dividi-las por 100 e adicionar o resultado a 1,01 para obter o expoente que deve ser usado.

Como exemplo, imagine que uma empresa está assumindo um projeto em um domínio no qual tem pouca experiência. O cliente do projeto não definiu o processo a ser usado e não permitiu um tempo no prazo de projeto para a análise de riscos. Uma nova equipe de desenvolvimento deve ser alocada para implementar esse sistema. A empresa estabeleceu recentemente um programa de melhoria de processo e foi avaliada como organização de Nível 2 de acordo com o modelo CMM (veja o Capítulo 28). Valores possíveis de avaliações usados no cálculo do expoente foram:

- *Precedência*. Este é um projeto novo para a organização — avaliação Baixo (4)
- *Flexibilidade do desenvolvimento*. Sem envolvimento de cliente — avaliação Muito alto (1)
- *Arquitetura/resolução de riscos*. Nenhuma análise de riscos realizada — avaliação Muito baixo (5)
- *Coesão da equipe*. Equipe nova, portanto sem informação — avaliação Nominal (3)
- *Maturidade do processo*. Algum controle de processo implantado — avaliação Nominal (3)

A soma desses valores é 16, de modo que você calcula o expoente adicionando 0,16 a 1,01, obtendo o valor de 1,17.

Os atributos (Tabela 26.8) usados para ajustar as estimativas iniciais e criar o multiplicador **M** no modelo de pós-arquitetura se dividem em quatro classes:

1. Atributos de produto se referem às características necessárias do produto de software em desenvolvimento.
2. Atributos de computador são restrições impostas sobre o software pela plataforma de hardware.
3. Atributos de pessoas são multiplicadores que levam em consideração a experiência e as capacitações das pessoas que trabalham no projeto.
4. Atributos de projeto se referem às características específicas do projeto de desenvolvimento de software.

O Quadro 26.1 mostra como esses direcionadores de custo influenciam as estimativas de esforço. Atribuí um valor para o expoente de 1,17, conforme explicado no exemplo anterior, e supus que RELY, CPLX, STOR, TOOL e SCED são os principais direcionadores de custo do projeto. Todos os outros direcionadores de custo têm valor nominal 1, portanto não afetam o cálculo do esforço.

No Quadro 26.1, atribuí valores máximos e mínimos para os principais direcionadores de custo para mostrar como eles influenciam na estimativa de esforço. Os valores considerados são os do manual de referência do COCOMO II (Boehm, 1997). Você pode ver que valores altos para os direcionadores de custo conduzem a uma estimativa de esforço que é mais de três vezes a estimativa inicial, enquanto os valores baixos reduzem a estimativa para aproximadamente um terço do original.

Tabela 26.7 Fatores de escala usados no cálculo de expoente do COCOMO II

Fator de escala	Explicação
Precedência	Reflete a experiência prévia da organização com esse tipo de projeto. Muito baixo significa nenhuma experiência prévia; extra alto significa que a organização está completamente familiarizada com esse domínio de aplicação.
Flexibilidade do desenvolvimento	Reflete o grau de flexibilidade no processo de desenvolvimento. Muito baixo significa que um processo prescrito é usado; extra alto significa que o cliente estabelece somente metas gerais.
Arquitetura/resolução de riscos	Reflete a extensão da análise de riscos realizada. Muito baixo significa pouca análise; extra alto significa uma análise completa e abrangente.
Coesão da equipe	Reflete o quanto os membros de equipe de desenvolvimento conhecem uns aos outros e trabalham juntos. Muito baixo significa interações muito difíceis; extra alto significa uma equipe integrada e eficaz, sem problemas de comunicação.
Maturidade do processo	Reflete a maturidade do processo da organização. O cálculo desse valor depende do Questionário de Maturidade de CMM, mas uma estimativa pode ser obtida subtraindo-se de 5 o nível de maturidade de processo CMM.

Tabela 26.8 Direcionadores de custos de projeto

Atributo	Tipo	Descrição
RELY	Produto	Confiabilidade necessária do sistema
CPLX	Produto	Complexidade dos módulos de sistema
DOCU	Produto	Extensão da documentação necessária
DATA	Produto	Tamanho da base de dados usada
RUSE	Produto	Percentual necessário de componentes reusáveis
TIME	Computador	Restrição de tempo de execução
PVOL	Computador	Volatilidade da plataforma de desenvolvimento
STOR	Computador	Restrições de memória
ACAP	Pessoal	Capacitação dos analistas de projetos
PCON	Pessoal	Continuidade de pessoal
PCAP	Pessoal	Capacitação do programador
PEXP	Pessoal	Experiência de programador no domínio de projeto
AEXP	Pessoal	Experiência do analista no domínio de projeto
LTEX	Pessoal	Experiência em linguagem e em ferramentas
TOOL	Projeto	Uso de ferramentas de software
SCED	Projeto	Compressão do prazo de desenvolvimento
SITE	Projeto	Extensão do trabalho em vários locais e qualidade das comunicações entre locais

Esses valores destacam as grandes diferenças entre tipos diferentes de projeto e as dificuldades de transferir experiência de um domínio de aplicação para outro.

As fórmulas propostas pelos desenvolvedores do modelo COCOMO II refletem sua experiência e dados, mas é um modelo extremamente complexo para compreender e usar. Existem muitos atributos e escopo considerável para incerteza na estimativa de seus valores. Em princípio, cada usuário deve ajustar o modelo e os valores de atributos de acordo com os seus próprios dados históricos de projeto, na medida em que isso refletirá as circunstâncias locais que afetam o modelo.

Quadro 26.1 Efeito dos direcionadores de custo nas estimativas de esforço

Valor do expoente	1,17
Tamanho do sistema (incluindo fatores de reuso e volatilidade de requisitos)	128.000 DSI
Estimativa inicial de COCOMO sem os direcionadores de custos	**730 pessoas-mês**
Confiabilidade	Muito alta, multiplicador = 1,39
Complexidade	Muito alta, multiplicador = 1,3
Restrição de memória	Alta, multiplicador = 1,21
Uso de ferramentas	Baixo, multiplicador = 1,12
Cronograma	Acelerado, multiplicador = 1,29
Estimativa de COCOMO ajustada	**2.306 pessoas-mês**
Confiabilidade	Muito baixa, multiplicador = 0,75
Complexidade	Muito baixa, multiplicador = 0,75
Restrição de memória	Nenhuma, multiplicador = 1
Uso de ferramentas	Muito alto, multiplicador = 0,72
Cronograma	Normal, multiplicador = 1
Estimativa de COCOMO ajustada	**295 pessoas-mês**

Na prática, contudo, poucas empresas têm coletado dados suficientes de projetos passados de maneira que apóie o ajuste do modelo. O uso prático do COCOMO II deve, portanto, iniciar com valores publicados para parâmetros de modelo, e é impossível para um usuário saber como eles se relacionam fielmente à sua própria situação. Isso significa que o uso prático do modelo COCOMO é limitado. Organizações muito grandes podem ter recursos para empregar um especialista em modelagem de custos para adaptar e usar os modelos do COCOMO II. Entretanto, para a maioria das empresas, o custo de calibração e o de aprendizagem para usar um modelo algorítmico como o COCOMO é tão alto que dificilmente elas vão introduzir essa abordagem.

26.3.2 Modelos algorítmicos de custos no planejamento de projeto

Um dos mais valiosos usos da modelagem algorítmica de custos é comparar diferentes maneiras de investir dinheiro para reduzir os custos de projeto. Isso é particularmente importante quando você precisa fazer um balanço de custos de hardware/software e pode ser necessário recrutar um novo pessoal com habilidades específicas de projeto. O modelo de código algorítmico o ajuda a avaliar os riscos de cada opção. A aplicação do modelo de custos revela a exposição financeira associada às diferentes decisões gerenciais.

Suponha um sistema embutido para controlar um experimento que será lançado no espaço. Os experimentos espaciais devem ser confiáveis e são sujeitos a rigorosos limites de peso. O número de chips em uma placa de circuitos deve ser minimizado. Em termos do modelo COCOMO, os multiplicadores baseados em restrições de computador e confiabilidade são maiores do que 1.

Existem três componentes para serem levados em consideração na elaboração de custos desse projeto:

1. O custo do hardware-alvo para executar o sistema
2. O custo da plataforma (computador mais software) para desenvolver o sistema
3. O custo do esforço necessário para desenvolver o software.

A Tabela 26.9 mostra algumas possíveis opções para esse projeto. Elas incluem gastar mais no hardware-alvo para reduzir custos de software ou investir em melhores ferramentas de desenvolvimento.

Os custos adicionais de hardware podem ser aceitáveis porque o sistema é um sistema especializado que não deve ser produzido em massa. Se o hardware é incorporado em produtos de consumo, contudo, investir em hardware-alvo para reduzir custos de software aumenta o custo por unidade do produto, independentemente do número vendido, que geralmente é indesejável.

A Tabela 26.9 mostra os custos de hardware, de software e o total para as opções A–F mostradas na Figura 26.3. A aplicação do modelo COCOMO II, sem os direcionadores de custo, prevê um esforço de 45 pessoas-mês para desenvolver um sistema de software embutido para essa aplicação. O custo médio para uma pessoa-mês de esforço é de 15 mil dólares.

Os multiplicadores relevantes baseiam-se nas restrições de armazenamento e de tempo de execução (**TIME** e **STOR**), na disponibilidade de apoio de ferramenta (*cross-compilers* etc.) para o sistema de desenvolvimento (**TOOL**), e na experiência da equipe de desenvolvimento em plataforma (**LTEX**). Em todas as opções, o multiplicador de confiabilidade (**RELY**) é 1,39, indicando que é necessário esforço extra significativo para desenvolver um sistema confiável.

Tabela 26.9 Custo de opções de gerenciamento

Opção	RELY	STOR	TIME	TOOLS	LTEX	Esforço total	Custo de software	Custo de hardware	Custo total
A	1,39	1,06	1,11	0,86	1	63	949393	100000	1049393
B	1,39	1	1	1,12	1,22	88	1313550	120000	1402025
C	1,39	1	1,11	0,86	1	60	895653	105000	1000653
D	**1,39**	**1,06**	**1,11**	**0,86**	**0,84**	**51**	**769008**	**100000**	**897490**
E	1,39	1	1	0,72	1,22	56	844425	220000	1044159
F	1,39	1	1	1,12	0,84	57	851180	120000	1002706

O custo de software (**SC**) é calculado conforme segue:

SC = Estimativa de esforço × RELY × TIME × STOR × TOOL × EXP × US$ 15.000

A opção A representa o custo de construir o sistema com apoio e pessoal existentes. Ela representa uma baseline para comparação. Todas as outras opções envolvem tanto mais gastos com hardware quanto o recrutamento (com custos e riscos associados) de um novo pessoal. A opção B mostra que a atualização de hardware não reduz necessariamente os custos. O pessoal apresenta falta de experiência com o novo hardware e, assim, o aumento no multiplicador de experiência não permite a redução nos multiplicadores **STOR** e **TIME**. É, realmente, mais adequado em custo aumentar memória do que toda a configuração do computador.

A opção D parece oferecer os menores custos para todas as estimativas básicas. Nenhuma despesa adicional de hardware é envolvida, mas um novo pessoal deve ser recrutado para o projeto. Se eles estiverem já disponíveis na empresa, é provavelmente a melhor opção a fazer. Se não, devem ser recrutados externamente, o que envolve custos e riscos significativos. Estes podem significar que as vantagens de custo dessa opção são muito menos significativas que o sugerido pela Tabela 26.9. A opção C oferece uma economia de 50 mil dólares com, virtualmente, nenhum risco associado. Gerentes de projeto conservadores selecionariam, provavelmente, essa opção em vez da opção D, que é de risco maior.

As comparações mostram a importância da experiência do pessoal como um multiplicador. Se pessoas de boa qualidade, com a experiência certa, são recrutadas, isso pode reduzir significativamente os custos de projeto. Isso é consistente com a explicação sobre fatores de produtividade na Seção 26.1. Ela revela que o investimento em hardware e ferramentas novas pode não ser adequado em custos. Alguns engenheiros podem preferir essa opção, pois ela lhes dá uma oportunidade para aprender e trabalhar com sistemas novos. Contudo, a perda de experiência é um efeito mais significativo no custo de sistema do que as economias que emergem do uso de um sistema de hardware novo.

26.4 Duração de projeto e seleção de pessoal

Além de estimar o esforço necessário para desenvolver um sistema de software e os custos gerais de projeto, os gerentes de projeto devem também estimar quanto tempo levará para desenvolver o software e quando o pessoal será necessário para trabalhar no projeto. O tempo de desenvolvimento de projeto é chamado de prazo de projeto. Cada vez mais, as empresas estão demandando prazos de desenvolvimento mais curtos de modo que os produtos sejam lançados para o mercado antes dos seus competidores.

O relacionamento entre o número de pessoas que trabalham em um projeto, o total de esforço necessário e o tempo de desenvolvimento não é linear. À medida que o pessoal aumenta, mais esforço pode ser necessário. A razão para isso é que as pessoas gastam mais tempo se comunicando e definindo interfaces entre as partes do sistema desenvolvidas por outras pessoas. Dobrar o número de pessoal (por exemplo) não significa, portanto, que a duração do projeto será reduzida à metade.

O modelo COCOMO inclui uma fórmula para estimar o tempo de calendário (**TDEV**) necessário para completar um projeto. A fórmula de cálculo de tempo é a mesma para todos os níveis de COCOMO:

$$\mathbf{TDEV} = 3 \times (\mathbf{PM})^{(0,33 + 0,2 * (B - 1,01))}$$

PM é o cálculo de esforço e **B** é o expoente calculado, conforme explicado anteriormente (**B** é 1 para o modelo de prototipação preliminar). Esse cálculo prevê o prazo nominal para o projeto.

Contudo, o prazo de projeto previsto e o prazo exigido pelo plano de projeto não são necessariamente a mesma coisa. O prazo planejado pode ser menor ou maior do que o prazo nominal previsto. Contudo, existe obviamente um limite de extensão de mudanças de prazos e o modelo COCOMO II prevê isso:

$$TDEV = 3 \times (PM)^{(0,33 + 0,2 * (B - 1,01))} \times SCEDPercentage/100$$

SCEDPercentage é o aumento ou diminuição percentual no prazo nominal. Se o valor previsto difere muito significativamente do prazo planejado, ele sugere que existe um alto risco de problemas de entrega de software, conforme planejado.

Para ilustrar o cálculo de prazo de desenvolvimento no COCOMO, suponha que são estimados 60 meses de esforço para desenvolver um sistema de software (opção C da Figura 26.3). Suponha que o valor do expoente B é 1,17. Da equação de prazo, o tempo necessário para completar o projeto é:

$$TDEV = 3 \times (60)^{0,36} = 13 \text{ meses}$$

Neste caso, não existe compressão ou expansão de prazo, de modo que o termo final na fórmula não tem efeito no cálculo.

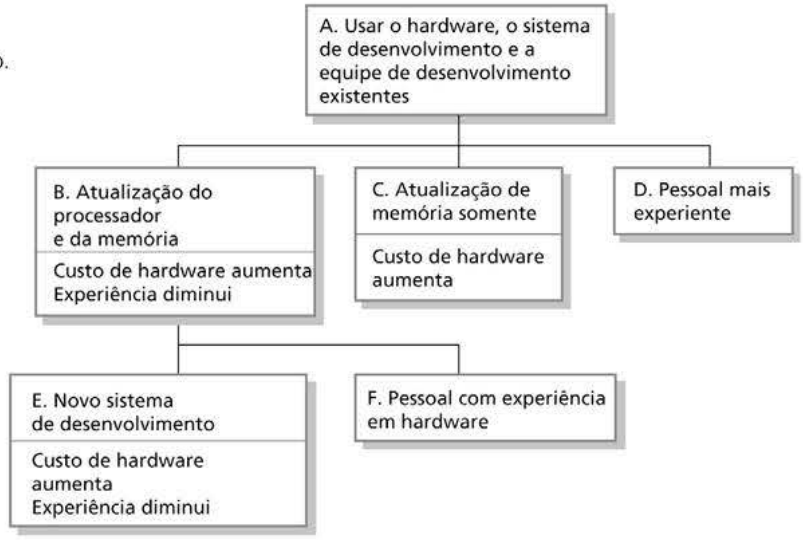
Uma conseqüência interessante do modelo COCOMO é que o tempo necessário para completar o projeto é uma função do esforço total necessário para o projeto. Ele não depende do número de engenheiros de software que trabalham no projeto. Isso confirma a noção de que a adição de mais pessoas em um projeto que está atrasado provavelmente não ajudará a recuperar o prazo. Myers (Myers, 1989) explica o problema da aceleração de prazo. Ele sugere que, provavelmente, os projetos podem enfrentar problemas significativos se tentarem desenvolver software sem darem tempo suficiente.

Dividir o esforço necessário para um projeto pelo prazo de desenvolvimento não fornece uma indicação útil do número de pessoas necessárias para a equipe de projeto. Geralmente, somente um número pequeno de pessoas é necessário no início de um projeto para produzir a especificação inicial. A equipe, então, é montada até atingir um pico durante o desenvolvimento e testes do sistema para, finalmente, o tamanho da equipe diminuir quando o sistema for preparado para a implantação. Uma montagem muito rápida do pessoal de projeto se mostra correlacionada com atrasos de prazo de projeto. Os gerentes de projetos devem, portanto, evitar a adição excessiva de pessoal ao projeto no início de seu ciclo de vida.

O aumento de esforço pode ser modelado pelo que é chamado de uma curva de Rayleigh (Londeix, 1987) e o modelo de estimativa de Putnam (Putnam, 1978), que incorpora um modelo de provisão de pessoal de projeto baseado nessas curvas. O modelo de Putnam também inclui o tempo de desenvolvimento como fator principal. Como o tempo de desenvolvimento é reduzido, o esforço necessário para desenvolver o sistema aumenta exponencialmente.

Figura 26.3

Opções de gerenciamento.



PONTOS-CHAVE

- Não existe, necessariamente, uma relação simples entre o preço cobrado por um sistema e seus custos de desenvolvimento. Fatores organizacionais podem significar que o preço orçado aumenta para compensar o aumento de risco ou a diminuição de ganho de vantagem competitiva.
- Os fatores que afetam a produtividade do software incluem a aptidão individual (o fator dominante), a experiência no domínio, o processo de desenvolvimento, o tamanho do projeto, o apoio de ferramentas e o ambiente de trabalho.
- O preço do software é freqüentemente calculado para obter um contrato, e a funcionalidade do sistema é, então, ajustada para atender ao preço estimado.

- Existem várias técnicas de estimativa de custo de software. Na preparação de uma estimativa, várias técnicas diferentes devem ser usadas. Se as estimativas divergirem muito, significa que estão disponíveis informações inadequadas.
- O modelo de custo COCOMO II é um modelo algorítmico de custos bem desenvolvido que leva em consideração atributos de projeto, de produto, de hardware e de pessoal ao formular uma estimativa de custos. Ele também inclui um meio para estimar prazos de desenvolvimento.
- Os modelos algorítmicos de custos podem ser usados para apoiar a análise de opções quantitativas. Eles permitem que o custo de várias opções possa ser calculado e, mesmo com erros, as opções possam ser comparadas objetivamente.
- O tempo necessário para completar um projeto não é simplesmente proporcional ao número de pessoas que trabalham nele. A adição de mais pessoas a um projeto atrasado pode aumentar, em vez de diminuir, o tempo necessário para terminá-lo.

LEITURAS SUGERIDAS

"Ten unmyths of project estimation". Artigo pragmático que explica as dificuldades práticas e desafios da estimativa de projeto, com algumas suposições fundamentais nessa área (P. Armour, *Comm. ACM*, 45(11), novembro de 2002).

Software cost estimation with COCOMO II. Este é um livro essencial sobre o modelo COCOMO II. Fornece uma descrição completa do modelo com muitos exemplos e inclui um software que implementa o modelo. É extremamente detalhado, e não é uma leitura leve. O artigo de Boehm citado a seguir é, em meu ponto de vista, uma introdução mais fácil ao modelo (B. Boehm, et al., 2000, Prentice Hall).

Software project management: readings and cases. Seleção de artigos e estudos de caso sobre gerenciamento de projeto de software o qual é particularmente forte ao abranger a modelagem algorítmica de custos (C. F. Kemerer (ed.), 1997, Irwin).

"Cost models for future software life cycle processes: COCOMO II". Introdução ao modelo de estimativa de custos COCOMO II, que inclui as razões para as fórmulas usadas. É mais fácil de ler do que o livro essencial (B. Boehm et al., *Annals of software engineering*, 1, Balzer Science Publishers, 1995).

EXERCÍCIOS

26.1 Sob quais circunstâncias uma empresa poderia cobrar um preço muito maior para um sistema de software do que o sugerido pela estimativa de custo mais a margem de lucro normal?

26.2 Descreva duas métricas usadas para medir a produtividade do programador. Comente, brevemente, sobre as vantagens e as desvantagens de cada uma dessas métricas.

26.3 No desenvolvimento de sistemas embutidos de tempo real de grande porte, sugira cinco fatores que provavelmente terão um efeito significativo na produtividade da equipe de desenvolvimento de software.

26.4 As estimativas de custos são inerentemente arriscadas, independentemente da técnica de estimativa usada. Sugira quatro maneiras pelas quais o risco pode ser reduzido em uma estimativa de custos.

26.5 Por que várias técnicas de estimativa devem ser usadas para produzir uma estimativa de custos de um sistema de software complexo e de grande porte?

26.6 Um gerente de software é responsável pelo desenvolvimento de um sistema de software crítico de segurança projetado para controlar uma máquina de radioterapia destinada a tratar pacientes que sofrem de câncer. Esse sistema é embutido na máquina e deve funcionar em um processador com propósito especial com uma quantidade fixa de memória (8 Mbytes). A máquina se comunica com um sistema de banco de dados para obter os detalhes do paciente e, depois do tratamento, registra automaticamente a dose de radiação emitida e outros detalhes de tratamento no banco de dados.

O método COCOMO é usado para estimar o esforço necessário para desenvolver esse sistema e é calculada uma estimativa de 26 pessoas-mês. Todos os multiplicadores de direcionadores de custo foram definidos como 1 quando a estimativa foi feita.

Explique por que essa estimativa deve ser ajustada para levar em consideração o projeto, as pessoas, o produto e os fatores organizacionais. Sugira quatro fatores que podem ter efeitos significativos sobre a estimativa inicial de COCOMO e proponha valores possíveis para esses fatores. Justifique por que você incluiu cada fator.

26.7 Forneça três razões por que as estimativas de custos algorítmicos preparados em diferentes organizações não são diretamente comparáveis.

26.8 Explique como a abordagem algorítmica para estimativa de custos pode ser usada pelos gerentes de projeto para a análise de opções. Sugira uma situação em que os gerentes possam escolher uma abordagem que não esteja baseada no menor custo de projeto.

26.9 Alguns projetos de software de grande porte envolvem a escrita de milhões de linhas de código. Sugira quão úteis são, provavelmente, os modelos de estimativas de custo para esses sistemas. Por que as suposições nas quais as estimativas estão baseadas podem ser inválidas para sistemas muito grandes?

26.10 É ético para uma empresa cotar um preço baixo para um contrato de software sabendo que os requisitos são ambíguos e que eles podem cobrar um preço alto pelas mudanças subseqüentes solicitadas pelo cliente?

26.11 A produtividade medida deve ser usada pelos gerentes durante o processo de avaliação de pessoal? Que cuidados são necessários para assegurar que a qualidade não será afetada por essa decisão?

27

Gerenciamento de qualidade

Objetivos

Os objetivos deste capítulo são apresentar o gerenciamento de qualidade e a medição de software. Após ler este capítulo, você:

- compreenderá o processo de gerenciamento de qualidade e as atividades centrais do processo de garantia de qualidade, de planejamento de qualidade e de controle de qualidade;
- compreenderá a importância dos padrões no processo de gerenciamento de qualidade;
- compreenderá o que são métricas de software e as diferenças entre métricas de predição e métricas de controle;
- compreenderá como a medição pode ser útil na avaliação de alguns atributos de qualidade de software;
- estará ciente das limitações atuais da medição de software.

Conteúdo

27.1 Qualidade de processo e de produto
27.2 Garantia de qualidade e padrões
27.3 Planejamento de qualidade
27.4 Controle de qualidade
27.5 Medições e métricas de software

A qualidade de software tem se aprimorado significativamente nos últimos 15 anos. Uma razão para isso é o fato de as empresas terem adotado novas técnicas e tecnologia, como o uso de desenvolvimento orientado a objetos e de ferramenta de apoio CASE associada. Além disso, contudo, tem havido uma conscientização maior da importância do gerenciamento de qualidade de software e da adoção de técnicas de gerenciamento de qualidade provenientes da manufatura de software.

Entretanto, qualidade de software é um conceito complexo que não é diretamente comparável com a qualidade na manufatura. Na manufatura, a noção de qualidade tem sido aquela em que o produto desenvolvido deve atender às suas especificações (Crosby, 1979). Em um mundo ideal essa definição deveria ser aplicada a todos os produtos, mas, para sistemas de software, há problemas com isso:

1. A especificação deve ser orientada para as características do produto que o cliente deseja. Entretanto, a organização que desenvolve pode também ter requisitos (como requisitos de facilidade de manutenção) que não estão incluídos na especificação.
2. Não sabemos como especificar certas características de qualidade (por exemplo, facilidade de manutenção) de maneira não ambígua.
3. Conforme explicado na Parte 1, que abordou engenharia de requisitos, é muito difícil escrever especificações de software completas. Portanto, embora um produto de software possa estar de acordo com a sua especificação, os usuários podem não considerá-lo como um produto de alta qualidade pelo fato de ele não corresponder às suas expectativas.

Você precisa reconhecer os problemas com especificações de software existentes e, portanto, com procedimentos de qualidade de projeto que não se apóiam em ter uma especificação perfeita. Em particular, atributos de software, como facilidade de manutenção, proteção ou eficiência não podem ser especificados explicitamente. Entretanto, eles têm um efeito grande na qualidade percebida no sistema. Explico esses atributos na Seção 27.3.

Algumas pessoas acham que a qualidade pode ser conseguida definindo-se padrões e procedimentos de qualidade organizacionais que verifiquem se esses padrões são seguidos pela equipe de desenvolvimento de software. Seu argumento é que os padrões devem englobar boas práticas; seguir essas boas práticas inevitavelmente conduz a produtos de alta qualidade. Na prática, entretanto, acredito que há muito mais gerenciamento de qualidade do que padrões e burocracia associada para assegurar que estes foram seguidos.

Bons gerentes de qualidade têm por objetivo desenvolver uma 'cultura de qualidade' na qual todos os responsáveis pelo desenvolvimento de produto estão comprometidos em atingir um alto nível de qualidade de produto. Eles encorajam as equipes a assumirem a responsabilidade pela qualidade de seus trabalhos e a desenvolverem novas abordagens para o aprimoramento da qualidade. Enquanto padrões e procedimentos são a base do gerenciamento de qualidade, os gerentes de qualidade experientes reconhecem que há aspectos intangíveis para a qualidade de software (elegância, capacidade de leitura etc.) que não podem ser incorporados em padrões. Eles apóiam as pessoas que estão interessadas nos aspectos intangíveis e encorajam o comportamento profissional em todos os membros da equipe.

O gerenciamento de qualidade formalizada é particularmente importante para equipes que estão desenvolvendo sistemas amplos e complexos. A documentação de qualidade é um registro do que tem sido feito por cada grupo no projeto. Isso ajuda as pessoas a verificar se tarefas importantes não foram esquecidas ou se uma parte da equipe não fez suposições incorretas sobre o que outras equipes fizeram. A documentação de qualidade é também um meio de comunicação ao longo da existência de um sistema. Isso permite que os grupos responsáveis pela evolução de sistema acompanhem o que a equipe de desenvolvimento tem feito.

Para sistemas menores, o gerenciamento de qualidade é ainda importante, mas uma abordagem mais informal pode ser adotada. Nem tanta papelada é necessária porque uma equipe pequena de desenvolvimento pode se comunicar informalmente. A questão principal de qualidade para desenvolvimento de sistemas pequenos é estabelecer uma cultura de qualidade e assegurar que todos os membros da equipe tenham uma abordagem positiva para qualidade de software.

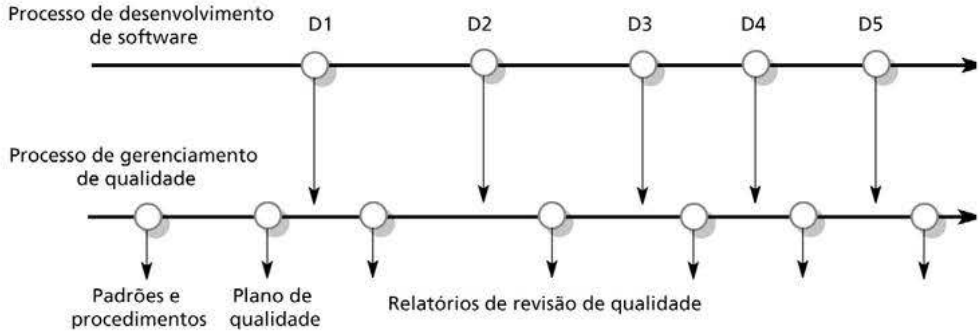
O gerenciamento de qualidade de software para sistemas de grande porte pode ser estruturado em três atividades principais:

1. *Garantia de qualidade*. Estabelecimento de um framework de procedimentos organizacionais e padrões que conduzem a um software de alta qualidade.
2. *Planejamento de qualidade*. Seleção de procedimentos e padrões apropriados deste framework, adaptados para um projeto de software específico.
3. *Controle de qualidade*. Definição e aprovação de processos que assegurem que a equipe de desenvolvimento de software tenha seguido os procedimentos e os padrões de qualidade de projeto.

A garantia de qualidade provê uma verificação independente no processo de desenvolvimento de software. O processo de gerenciamento de qualidade verifica as entregas de projeto para assegurar que elas estejam consistentes com os padrões e objetivos organizacionais (Figura 27.1). A equipe de garantia de qualidade deve ser independente da equipe de desenvolvimento de modo que possam ter uma visão objetiva do software. Ela relata os problemas e as dificuldades para a gerência sênior da organização.

Uma equipe independente deve ser responsável pelo gerenciamento de qualidade e deve se reportar à gerência acima do nível de gerente de projeto. A equipe de gerenciamento de qualidade não deve estar associada a algum grupo de desenvolvimento específico, mas deve ter uma responsabilidade ampla no âmbito organizacional pelo gerenciamento de qualidade. A razão para isso é que os gerentes têm de manter o orçamento e o cronograma do projeto. Se surgirem problemas, eles podem ficar tentados a comprometer a qualidade do produto de modo a cumprir o cronograma. Uma equipe de gerenciamento de qualidade independente assegura que os objetivos organizacionais de qualidade não sejam comprometidos por orçamentos de curto prazo e considerações de cronograma.

Figura 27.1 Gerenciamento de qualidade e desenvolvimento de software.



27.1 Qualidade de processo e de produto

A suposição fundamental da gerência de qualidade é que a qualidade do processo de desenvolvimento afeta diretamente a qualidade dos produtos entregues. Essa suposição provém de sistemas de manufatura, na qual a qualidade de produto está intimamente relacionada ao processo de produção. Em um sistema de manufatura automatizada, o processo abrange configuração, estabelecimento e operação de máquinas envolvidas no processo. Uma vez que as máquinas estejam operando corretamente, a qualidade de produto flui naturalmente. Você deve medir a qualidade do produto e mudar o processo até o nível de qualidade que você precisa que seja atingido. A Figura 27.2 ilustra essa abordagem baseada em processos para se atingir a qualidade de produto.

Há uma ligação nítida entre processo e qualidade de produto na manufatura, porque o processo é relativamente fácil de padronizar e monitorar. Uma vez que os sistemas de manufatura estejam calibrados, eles podem ser rodados repetidamente para sair produtos de alta qualidade. Entretanto, o software não é manufaturado, mas projetado. O desenvolvimento de software é um processo mais criativo do que mecânico; dessa maneira a influência de habilidades individuais e de experiência é significativa. Fatores externos, como a novidade de uma aplicação ou a pressão comercial para um release de produto antecipado, também afetam a qualidade de produto independentemente do processo usado.

No desenvolvimento de software, portanto, o relacionamento entre qualidade de processo e qualidade de produto é mais complexo. É difícil medir os atributos de qualidade de software, como facilidade de manutenção, mesmo após o uso do software por um longo período. Conseqüentemente, é difícil dizer como as características de processo influenciam esses atributos. Além disso, por causa do papel do projeto e da criatividade no processo de software, você não pode prever como as mudanças de processo influenciarão na qualidade de produto. Entretanto, a experiência tem mostrado que a qualidade de processo tem uma influência significativa na qualidade do software. O gerenciamento de qualidade de processo e o aprimoramento podem certamente conduzir a poucos defeitos no software entregue.

O gerenciamento de qualidade de processo envolve:

1. Definição de padrões de processo, como 'como' e 'quando' as revisões devem ser conduzidas.
2. Monitoração do processo de desenvolvimento para assegurar que os padrões estão sendo seguidos.
3. Relato do processo de software para a gerência de projeto e para o comprador do software.

Um problema com a garantia de qualidade baseada em processos é que a equipe de garantia de qualidade (QA — *Quality Assurance*) pode insistir em que os processos padrões devem ser usados independentemente do tipo de software em desenvolvimento. Por exemplo, os padrões de qualidade de processo para sistemas críticos podem definir que a especificação deve estar completa e aprovada antes de se iniciar a implementação. Entretanto, alguns sistemas críticos podem requerer

Figura 27.2 Qualidade baseada em processo.

prototipação, na qual os programas são implementados sem uma especificação completa. Passei por situações em que a equipe de gerenciamento de qualidade sugeriu que esse protótipo não deve ser realizado porque a qualidade do protótipo não pode ser monitorada. Nessas situações, a gerência sênior precisa intervir para assegurar que o processo de qualidade apóie, e não impeça, o desenvolvimento do produto.

27.2 Garantia de qualidade e padrões

Garantia de qualidade é o processo de definição de como a qualidade de software pode ser atingida e como a organização de desenvolvimento sabe que o software possui o nível de qualidade necessário. Como sugeri anteriormente, o processo QA está, principalmente, relacionado à definição e seleção de padrões que devem ser aplicados ao processo de desenvolvimento de software ou ao produto de software. Como parte do processo de QA, você pode selecionar e adquirir ferramentas e métodos para apoiar esses padrões.

Os dois tipos de padrões que podem ser estabelecidos como parte do processo de garantia de qualidade são:

1. *Padrões de produto*. Esses padrões se aplicam ao produto de software em desenvolvimento. Eles incluem padrões de documentos, como a estrutura de documentos de requisitos; padrões de documentação, como um cabeçalho de comentário padronizado para uma definição de classe de objeto; e padrões de codificação, que definem como uma linguagem de programação deve ser usada.
2. *Padrões de processo*. Esses padrões definem os processos que devem ser seguidos durante o desenvolvimento de software. Podem incluir definições de processos de especificação, projeto e de validação, e uma descrição dos documentos que devem ser escritos ao longo destes processos.

Conforme sugeri na Seção 27.1, há uma estreita ligação entre padrões de produto e de processo. Os padrões de produto aplicam-se à saída do processo de software e, em muitos casos, os padrões de processo incluem atividades de processo específicas que asseguram que os padrões de produto sejam seguidos.

Os padrões de software são importantes por várias razões:

1. São baseados no conhecimento sobre as melhores e as mais apropriadas práticas para a empresa. Esse conhecimento freqüentemente é adquirido somente após um grande número de tentativas e erros. Constituí-lo em um padrão ajuda a empresa a evitar a repetição de erros cometidos no passado. Os padrões capturam a sabedoria valiosa para a organização.
2. Eles provêem um framework para a implementação do processo de garantia de qualidade. Considerando que esses padrões englobam as melhores práticas, a garantia da qualidade envolve assegurar que padrões apropriados foram selecionados e são usados.
3. Eles ajudam na continuidade, isto é, quando o trabalho realizado por uma pessoa é assumido e continuado por uma outra pessoa. Os padrões asseguram que todos os engenheiros dentro de uma organização adotem as mesmas práticas. Conseqüentemente, o esforço de aprendizado é reduzido quando se inicia um novo trabalho.

O desenvolvimento de padrões de projeto de engenharia de software é um processo difícil e demorado. Instituições nacionais e internacionais, como o US DoD (*United States Department of Defense*), ANSI (*American National Standards Institute*), BSI (*British Standards Institution*), NATO (*North Atlantic Treaty Organization*) e a IEEE (*Institute of Electrical and Electronic Engineers*) têm sido ativas na produção de padrões. Trata-se de padrões gerais que podem ser aplicados em uma variedade de projetos. Instituições como a NATO e outras organizações de defesa podem requerer que seus próprios padrões sejam seguidos nos contratos de software.

Padrões nacionais e internacionais foram desenvolvidos abrangendo a terminologia de engenharia de software, linguagens de programação, como Java e C++, notações, como símbolos gráficos, procedimentos para derivar e escrever requisitos de software, procedimentos de garantia de qualidade e processos de verificação e validação de software (IEEE, 2003).

As equipes de garantia de qualidade que estão desenvolvendo padrões para uma empresa devem, normalmente, basear seus padrões organizacionais em padrões nacionais e internacionais. Usando esses padrões como ponto de partida, a equipe de garantia de qualidade deve elaborar um 'manual' de padrões. Esse manual deve definir os padrões necessários para sua organização. Exemplos de padrões que poderiam ser incluídos no manual são mostrados na Tabela 27.1.

Os engenheiros de software consideram, algumas vezes, que os padrões sejam burocráticos e irrelevantes para a atividade técnica de desenvolvimento de software. Isso é de certa maneira provável quando os padrões exigem preenchimentos de tediosos formulários e registros de trabalho. Embora concordem geralmente sobre a necessidade geral de padrões, os engenheiros, freqüentemente, encontram boas razões pelas quais os padrões não são necessariamente apropriados para seu projeto especificamente.

Tabela 27.1 Padrões de produto e de processo

Padrões de produto	Padrões de processo
Formulário de revisão de projeto	Conduta de revisão de projeto
Estrutura de documento de requisitos	Envio do documento para o CM
Formato do cabeçalho de método	Processo de liberação de versão
Estilo de programação em Java	Processo de aprovação de plano de projeto
Formato de plano de projeto	Processo de controle de mudança
Formulário de solicitação de mudança	Processo de registro de teste

Para evitar esses problemas, os gerentes de qualidade, que estabelecem os padrões, precisam estar adequadamente apoiados e devem seguir os seguintes passos:

1. Envolver engenheiros de software na seleção de padrões de produto. Eles devem compreender por que os padrões foram projetados e, assim, provavelmente, ficarão mais comprometidos com esses padrões. O documento de padrões não deve simplesmente estabelecer um padrão a ser seguido, mas deve incluir as razões pelas quais as decisões de padronização foram tomadas.
2. Revisar e modificar os padrões regularmente para refletir as mudanças de tecnologia. Uma vez que os padrões são desenvolvidos, eles tendem a ser preservados em um manual de padrões da empresa, e a gerência, freqüentemente, reluta em modificá-los. Um manual de padrões é essencial, mas ele deve evoluir para refletir as mudanças de circunstâncias e de tecnologia.
3. Prover ferramentas de software para apoiar os padrões, sempre que possível. Os padrões que envolvem muitos registros são as causas de muitas reclamações devido ao trabalho tedioso envolvido em sua implementação. Se o apoio de ferramentas estiver disponível, você não necessita de esforço adicional para seguir os padrões de desenvolvimento de software.

Os padrões de processo podem causar dificuldades se um processo não prático é imposto para a equipe de desenvolvimento. Diferentes tipos de software necessitam de diferentes processos de desenvolvimento. Não existe uma razão para prescrever uma maneira específica de trabalho se ele for inapropriado para um projeto ou para uma equipe de projeto. Cada gerente de projeto deve então ter a autoridade de modificar os padrões de processo de acordo com as circunstâncias individuais. Contudo, os padrões que se relacionam à qualidade de produto e ao processo pós-entrega devem ser somente alterados depois de considerações cuidadosas.

O gerente de projeto e o gerente de qualidade podem evitar os problemas de padrões inapropriados por meio de um planejamento cuidadoso de qualidade no início do projeto. Eles devem decidir quais dos padrões constantes no manual devem ser usados sem mudanças, quais devem ser modificados e quais devem ser ignorados. Novos padrões talvez possam ser necessários em resposta a determinado requisito de projeto. Por exemplo, padrões para especificações formais podem ser necessários se não foram usados em projetos anteriores. À medida que a equipe ganha experiência com eles, você deve planejar a modificação e a extensão desses novos padrões.

27.2.1 ISO 9000

Um conjunto internacional de padrões que pode ser usado no desenvolvimento de um sistema de gerenciamento de qualidade em todas as indústrias é chamado de ISO 9000. Os padrões ISO 9000 podem ser aplicados para uma variedade de organizações, desde a indústria de manufatura até a indústria de serviço. O padrão ISO 9001 é o mais geral desses padrões e se aplica às organizações que se dedicam a processos de qualidade nas organizações que projetam, desenvolvem e mantêm produtos. Um documento de apoio (ISO 9000-3) interpreta o ISO 9001 para o desenvolvimento de software. Vários livros que descrevem o padrão ISO 9001 estão disponíveis (Johnson, 1993; Oskarsson e Glass, 1995; Peach, 1996; Bamford e Deibler, 2003).

O padrão ISO 9001 não é especificamente voltado para desenvolvimento de software, mas estabelece princípios gerais que podem ser aplicados ao software. O padrão ISO 9001 descreve vários aspectos do processo de qualidade e exibe os padrões e os procedimentos organizacionais que a empresa deve definir. Eles devem ser documentados em um manual de qualidade da organização. A definição de processo deve incluir descrições da documentação necessária para demonstrar que os processos definidos foram seguidos durante o desenvolvimento do produto.

O padrão ISO 9001 não define os processos de qualidade que devem ser usados. De fato, ele não restringe de modo nenhum os processos usados em uma organização. Isso permite flexibilidade por meio de setores industriais e significa que pequenas empresas podem ter processos não burocráticos e, ainda assim, estar em conformidade com a ISO 9000. Contudo, essa flexibilidade significa que você não pode fazer quaisquer suposições sobre a similaridade ou a diferença entre os processos nas empresas em conformidade ao ISO 9000.

A Tabela 27.2 mostra as áreas cobertas no padrão ISO 9001. Não tenho espaço nesta seção para explicar esse padrão em profundidade. Ince (Ince, 1994) e Oskarsson e Glass (Oskarsson e Glass, 1995) dão mais detalhes sobre como o padrão pode ser usado para desenvolver processos de gerenciamento de qualidade de software. Os relacionamentos entre o ISO 9000, o manual de qualidade e os planos de qualidade de projeto individual são mostrados na Figura 27.3. Derivei essa figura de um modelo apresentado no livro de Ince (Ince, 1994).

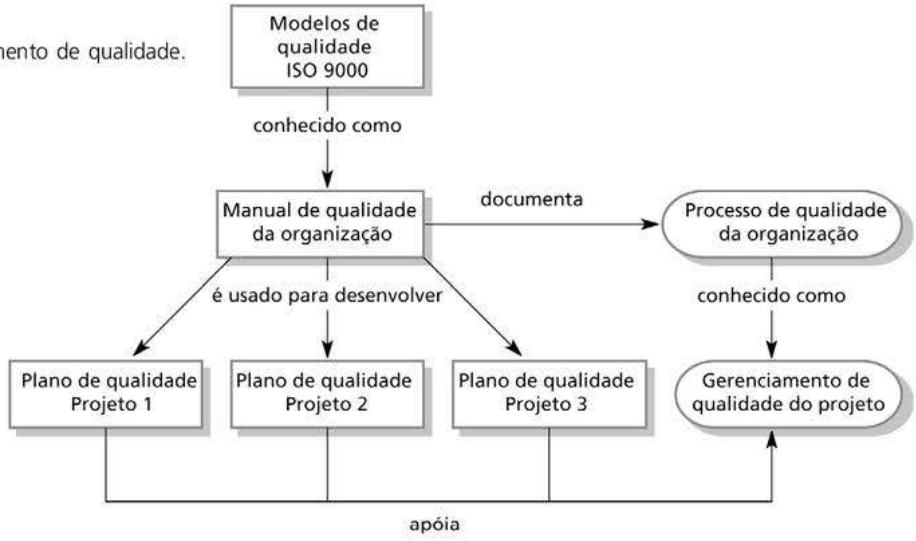
Os procedimentos de garantia de qualidade em uma organização são documentados em um manual que define o processo de qualidade. Em alguns países, as autoridades certificadoras certificam que o processo de qualidade, conforme expresso no manual de qualidade, está em conformidade ao padrão ISO 9001. Cada vez mais, os clientes procuram pela certificação ISO 9000 de um fornecedor como um indicador de com que seriedade esse fornecedor trata a qualidade.

Algumas pessoas pensam que a certificação ISO 9000 significa que a qualidade do software produzido pelas empresas certificadas será melhor do que a das empresas não certificadas. Esse certamente não é o caso. O ISO 9000 está simplesmente relacionado com a definição de processos que serão usados em uma empresa e a documentação associada, como processos de controle que podem explicitamente mostrar que esses processos foram seguidos. Isso não está relacionado com a garantia de que os processos refletem as melhores práticas ou com a qualidade de produto.

Tabela 27.2 Áreas cobertas pelo modelo ISO 9001 para garantia de qualidade.

Responsabilidade da gerência	Sistema de qualidade
Controle de produtos que estão em não-conformidade	Controle de projeto
Manuseio, armazenamento, empacotamento e entrega	Compras
Produtos fornecidos para o comprador	Identificação e rastreabilidade de produto
Controle de processo	Inspeção e teste
Equipamento de inspeção e teste	Status de inspeção e teste
Revisão de contrato	Ação corretiva
Controle de documento	Registros de qualidade
Auditorias internas de qualidade	Treinamento
Prestação de serviço	Técnicas estatísticas

Figura 27.3
ISO 9000 e gerenciamento de qualidade.



Portanto, uma empresa poderia definir procedimentos de teste de produto (digamos) que conduzem a testes de software incompletos. Assim, à medida que esses processos forem seguidos e documentados, a empresa estaria seguindo o padrão ISO 9001. Enquanto essa situação for indesejável, não há dúvida de que alguns padrões de empresas serão muito inadequados e trarão pouca contribuição para a real qualidade de software.

27.2.2 Padrões de documentação

Padrões de documentação em um projeto de software são importantes porque os documentos são o único modo tangível de representação do software e do processo de software. Os documentos padronizados têm aparência, estrutura e qualidade consistentes e devem, portanto, ser mais fáceis de ler e de compreender.

Existem três tipos de padrões de documentação:

1. *Padrões de processo de documentação*. Esses padrões definem o processo que deve ser seguido para a produção de documentos.
2. *Padrões de documentos*. Esses padrões regem a estrutura e a apresentação de documentos.
3. *Padrões de intercâmbio de documentos*. Esses padrões asseguram que todas as cópias eletrônicas de documentos sejam compatíveis.

Os padrões de processo de documentação definem o processo para produzir documentos. Isso significa que você deve estabelecer procedimentos envolvidos no desenvolvimento de documentos e as ferramentas de software usadas para a produção de documentos. Você deve definir procedimentos de verificação e de refinamento para assegurar que documentos de alta qualidade são produzidos.

Os padrões de qualidade de processo de documentação devem ser flexíveis e capazes de lidar com todos os tipos de documentos. Para documentos de trabalho ou memorandos eletrônicos, não existe uma necessidade explícita para verificação de qualidade. Contudo, para documentos formais — ou seja, os que serão usados para desenvolvimento posterior ou liberados para os clientes — você deve usar um processo de qualidade formal. A Figura 27.4 é um modelo de um suposto processo de documentação.

Criar, verificar, revisar e refazer o rascunho é um processo iterativo. Deve ser contínuo até que um documento com qualidade aceitável seja produzido. O nível de qualidade aceitável depende do tipo de documento e os potenciais leitores do documento.

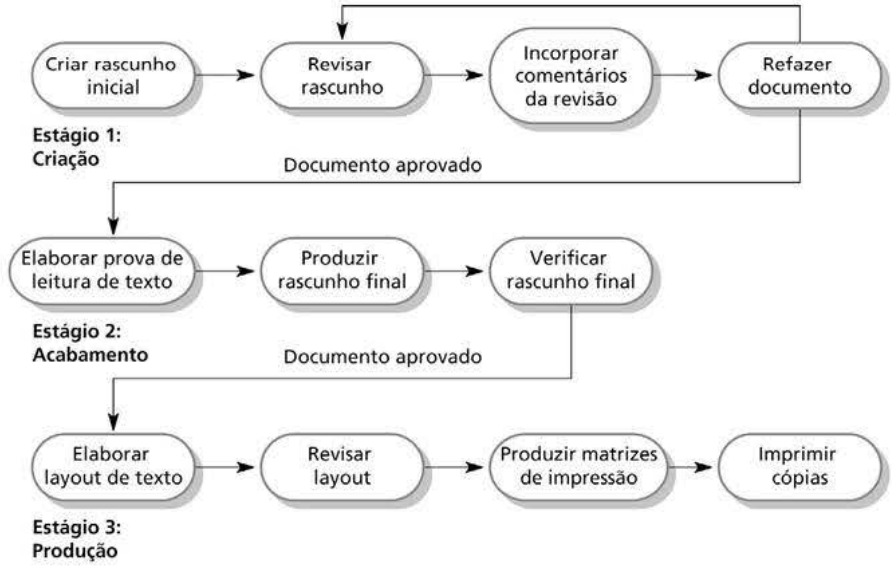
Os padrões de documentos devem se aplicar a todos os documentos produzidos durante um projeto de desenvolvimento de software. Os documentos devem ter estilo consistente e boa aparência, e os documentos de mesmo tipo devem ter uma estrutura consistente. Embora os padrões de documentos devam ser adaptados às necessidades de um projeto específico, é uma boa prática usar um mesmo 'estilo' para todos os documentos produzidos por uma organização.

Exemplos de padrões de documentos que podem ser desenvolvidos são:

1. *Padrões de identificação de documentos*. Como os grandes projetos de desenvolvimento de sistema podem produzir milhares de documentos, cada documento deve ser unicamente identificado. Para documentos formais, esse identifi-

Figura 27.4

Processo de produção de documentos incluindo verificação de qualidade.



cador pode ser o identificador formal definido pelo gerente de configuração. Para documentos informais, o gerente de projeto pode definir a forma do documento.

2. *Padrões de estrutura de documentos*. Cada classe de documentos produzidos durante um projeto de software deve seguir alguma estrutura padrão. Os padrões de estrutura devem definir as seções a serem incluídas e devem especificar as convenções usadas para numeração de páginas, informações de cabeçalho e de rodapé de página e numeração de seção e subseção.
3. *Padrões de apresentação de documentos*. Os padrões de apresentação de documentos definem um 'estilo' para os documentos e contribuem significativamente para a consistência dos documentos. Eles incluem a definição de fontes e de estilos usados nos documentos, o uso de logotipos e nomes de empresas, o uso de cores para destacar a estrutura de documentos, e assim por diante.
4. *Padrões de atualização de documentos*. Quando um documento evolui para refletir as mudanças no sistema, deve ser usada uma maneira consistente para indicar as mudanças de documentos. Você pode usar cores de capa para indicar a versão do documento e barras de mudanças na margem para indicar os parágrafos modificados ou adicionados. Contudo, recomendo que não se use o acompanhamento de mudanças presente em alguns processadores de texto comumente usados. Se houver mais do que dois autores, penso que o acompanhamento de mudanças torna-se confuso em vez de útil.

Os padrões para intercâmbio de documentos são importantes quando cópias eletrônicas de documentos são trocadas. O uso de padrões de intercâmbio permite que os documentos sejam transferidos eletronicamente e sejam recriados na sua forma original.

Supondo que o uso de ferramentas padronizadas seja obrigatório nos padrões de processo, os padrões de intercâmbio definem as convenções para o uso dessas ferramentas. Exemplos de padrões de intercâmbio incluem o uso de uma folha de estilo padrão, se um processador de texto é usado, ou limitações de uso de macros de documento para evitar uma possível infecção de vírus. Os padrões de intercâmbio podem também limitar os estilos de fontes e de texto usados devido às capacidades diferentes de impressoras e displays.

27.3 Planejamento de qualidade

Planejamento de qualidade é o processo de desenvolvimento de um plano de qualidade para um projeto. O plano de qualidade deve estabelecer as qualidades de software desejadas e descrever como elas devem ser avaliadas. O plano define, portanto, o que significa 'alta qualidade' de software. Sem essa definição, os engenheiros podem fazer suposições diferentes e, às vezes, conflitantes sobre quais atributos de produto devem ser otimizados.

O plano de qualidade deve selecionar os padrões organizacionais apropriados para determinado produto e o processo de desenvolvimento. Novos padrões podem ser definidos se o projeto usar novos métodos e ferramentas. Humphrey (Humphrey, 1989), em seu livro clássico sobre gerenciamento de software, sugere uma estrutura geral para um plano de qualidade. Ela inclui:

1. *Apresentação do produto*. Descrição do produto, o mercado pretendido e as expectativas de qualidade para o produto.
2. *Planos de produto*. As datas críticas de liberação e as responsabilidades para o produto junto com planos para serviços de distribuição e de produto.
3. *Descrições de processo*. Os processos de desenvolvimento e de serviços devem ser usados para o desenvolvimento e gerenciamento de produto.
4. *Metas de qualidade*. As metas e os planos de qualidade para o produto incluindo identificação e justificativa de atributos críticos de qualidade de produto.
5. *Riscos e gerenciamento de riscos*. Os riscos principais que poderiam afetar a qualidade de produto e as ações para tratar esses riscos.

Os planos de qualidade, obviamente, diferem em detalhes dependendo do tamanho e do tipo de sistema em desenvolvimento. Contudo, quando se escrevem planos de qualidade, você deve tentar mantê-los tão breves quanto possível. Se o documento for muito longo, as pessoas não o lerão e isso anulará o propósito de produzir um plano de qualidade.

Existe grande variedade de atributos de qualidade de softwares potenciais (Quadro 27.1) que você deve considerar durante o processo de planejamento de qualidade. Em geral, não é possível para nenhum sistema ser otimizado para todos os atributos. No plano de qualidade, você deve, então, definir os atributos de qualidade mais importantes para o software em desenvolvimento. Pode ser que a eficiência seja crítica e outros fatores devam ser sacrificados para atingir tal meta. Se você estabeleceu isso no plano de qualidade, os engenheiros que trabalham no desenvolvimento podem cooperar para atingir essa

Quadro 27.1
Atributos de qualidade de software.

Segurança	Facilidade de compreensão	Portabilidade
Proteção	Facilidade de testes	Facilidade de uso
Confiabilidade	Adaptabilidade	Facilidade de reuso
Facilidade de recuperação	Modularidade	Eficiência
Robustez	Complexidade	Facilidade de aprendizado

eficiência. O plano deve incluir também uma definição do processo de avaliação de qualidade. Essa deve ser uma maneira padronizada de avaliação se certa qualidade, como facilidade de manutenção ou robustez, estiver presente no produto.

27.4 Controle de qualidade

Controle de qualidade envolve a monitoração do processo de desenvolvimento de software para assegurar que os procedimentos e os padrões de garantia de qualidade estão sendo seguidos. Conforme explicado anteriormente neste capítulo (veja a Figura 27.1), os produtos do processo de software são verificados contra os padrões de projeto definidos no processo de controle de qualidade.

Existem duas abordagens complementares que podem ser usadas para verificar a qualidade dos produtos de projeto:

1. Revisões de qualidade nas quais o software, sua documentação e os processos usados para produzir o software são revisados por um grupo de pessoas. A revisão é responsável por verificar que os padrões de projeto foram seguidos e que o software e os documentos estão em conformidade com esses padrões. Os desvios dos padrões são anotados e o gerente de projeto é alertado para eles.
2. Avaliação automatizada de software na qual o software e os documentos produzidos são processados por algum programa e comparados com padrões que se aplicam naquele projeto de desenvolvimento específico. Essa avaliação automatizada pode envolver medições de alguns atributos de software e comparações das medições com algum nível desejável. Explico medições e métricas de software na Seção 27.5.

27.4.1 Revisões de qualidade

As revisões são métodos de validação de qualidade de um processo ou produto amplamente usados. Envolvem um grupo de pessoas que examina parte ou o todo de um processo de software, sistema ou sua documentação associada para descobrir problemas potenciais. As conclusões da revisão são formalmente registradas e passadas para o autor ou quem mais for o responsável por corrigir os problemas detectados.

A Tabela 27.3 descreve vários tipos de revisão, entre eles revisões para gerenciamento de qualidade. As inspeções de programa já foram abordadas no Capítulo 22. As revisões de progresso são parte do processo de gerenciamento, conforme explicado no Capítulo 5. Os processos de revisão têm muito em comum; já descrevi o processo de preparação de uma revisão no Capítulo 22.

O propósito da equipe de revisão é detectar erros e inconsistências, e apontá-los para o projetista ou o autor do documento. As revisões são baseadas em documentos e não são limitadas a especificações, projetos ou código. Os documentos, como

Tabela 27.3 Tipos de revisão

Tipo de revisão	Propósito principal
Inspeções de projeto ou de programa	Detectar erros detalhados nos requisitos, projeto ou código. Uma lista de verificação de possíveis erros deve guiar a revisão.
Revisões de progresso	Fornecer informações para a gerência sobre o progresso geral do projeto. Esta é uma revisão de processo e de produto e concentra-se em custos, planejamentos e prazos.
Revisões de qualidade	Conduzir uma análise técnica dos componentes de produto ou documentação para encontrar inconsistências entre especificação e projeto, código ou documentação de componente e assegurar que padrões de qualidade definidos foram seguidos.

modelos de processos, planos de teste, procedimentos de gerenciamento de configuração, padrões de processo e manuais de usuário podem todos ser revisados.

A equipe de revisão deve ter um núcleo de três a quatro pessoas selecionadas como revisores principais. Um membro deve ser um projetista sênior que possa assumir a responsabilidade de tomar as decisões técnicas significativas. Os principais revisores podem convidar outros membros de projeto, como os projetistas dos subsistemas, para contribuir com a revisão. Eles podem não estar envolvidos na revisão de todo o documento. Ou seja, eles se concentram naquelas partes que afetam o seu trabalho. Alternativamente, a equipe de revisão pode circular o documento em revisão e solicitar comentários escritos para uma variedade maior de membros de projeto.

Os documentos a serem revisados devem ser distribuídos bem antecipadamente da revisão para permitir que os revisores tenham tempo para lê-los e compreendê-los. Embora esse atraso possa interromper o processo de desenvolvimento, a revisão não será eficaz se a equipe de revisão não tiver uma compreensão adequada dos documentos antes que a revisão ocorra.

A revisão em si deve ser relativamente curta (duas horas, no máximo). O autor do documento em revisão deve 'percorrer' o documento com a equipe de revisão. Um membro da equipe deve moderar a revisão e um outro deve registrar formalmente as decisões e as ações a serem tomadas. Durante a revisão, o moderador é responsável por garantir que todos os comentários sejam considerados. O moderador da revisão deve assinar um registro de comentários e as ações acordadas durante a revisão. Esse registro é então arquivado como parte da documentação formal de projeto. Se somente problemas menores forem descobertos, uma revisão posterior pode não ser necessária. O moderador é responsável por assegurar que as mudanças necessárias foram feitas. Se mudanças maiores forem necessárias, uma revisão de acompanhamento pode ser organizada.

27.5 Medições e métricas de software

As revisões de software são onerosas, demoradas e, inevitavelmente, atrasam a conclusão de um sistema de software. Idealmente, seria possível acelerar o processo de revisão por meio de ferramentas para processar o projeto ou programa de software e fazer algumas avaliações automatizadas da qualidade de software. Essas avaliações podem verificar que o software atingiu o limiar de qualidade necessária e, quando isso não ocorrer, destacar essas áreas que a revisão deve enfocar.

A medição de software se dedica a derivar um valor numérico para algum atributo de um produto de software ou de um processo de software. Comparando-se esses valores uns com os outros e aos padrões que se aplicam em uma organização, você pode ser capaz de tirar conclusões sobre a qualidade de software ou dos processos de software. Por exemplo, digamos que uma organização planeja introduzir uma nova ferramenta de teste de software. Antes de introduzir a ferramenta, você registra o número de defeitos de software descobertos em um dado tempo; depois da introdução da ferramenta, você repete esse processo. Se mais defeitos forem descobertos no mesmo intervalo de tempo, depois da introdução da ferramenta, você pode concluir que ela fornece apoio útil ao processo de validação de software.

Existem duas maneiras nas quais as medições de software podem ser usadas:

1. *Para fazer previsões gerais sobre um sistema*. Ao medir as características dos componentes de sistema e, em seguida, agregar essas medições, você pode derivar uma estimativa geral de algum atributo de sistema, como o número de defeitos no sistema.
2. *Para identificar componentes anômalos*. As medições podem identificar componentes individuais cujas características desviem de alguma regra. Por exemplo, você pode medir os componentes para descobrir os com maior complexidade e, supondo que esses são provavelmente os que terão mais erros, concentrar-se sobre esses componentes durante o processo de revisão.

Uma métrica de software é qualquer tipo de medição que se referia a um sistema de software, processo ou documentação relacionada. Exemplos são: medidas de tamanhos de um produto em linhas de código; índice de *Fog* (Gunning, 1962), que é uma medida de facilidade de leitura de uma passagem de um texto escrito; o número de defeitos relatados em um produto de software entregue; o número de pessoas-dia necessário para desenvolver um componente de sistema.

Uma série de grandes empresas, como a Hewlett-Packard (Grady, 1993), AT&T (Barnard e Price, 1994) e Nokia (Kilpi, 2001), introduziu programas de métricas e estão usando métricas coletadas em seus processos de gerenciamento de qualidade. O maior foco tem sido a coleta de métricas sobre defeitos de programas e os processos de verificação e validação. Offen e Jeffrey (Offen e Jeffrey, 1997) e Hall e Fenton (Hall e Fenton, 1997) explicam a introdução de programas de métrica na indústria. O manual denominado ami (aplicação de métricas na indústria) (Pulford, et al., 1996) fornece orientações detalhadas sobre medições e uso de medições para o aprimoramento do processo.

Contudo, a maioria de empresas ainda não usa medições sistemáticas para avaliar a qualidade de software. Uma razão é que, em muitas empresas, os processos de software são precariamente definidos e controlados, e não são suficientemente

maduros para se fazer uso de medições. Outra razão é que não existem padrões para métricas e, por isso, existe um apoio limitado de ferramentas para coleta e análise de dados. Muitas empresas não estarão preparadas para introduzir medições até que esses padrões e ferramentas estejam disponíveis.

As métricas de software podem ser tanto métricas de controle ou de predição. Ambas, métricas de controle e de predição, podem influenciar na tomada de decisão de gerenciamento, conforme mostrado na Figura 27.5. As métricas de controle são usualmente associadas com processos de software; as métricas de predição são associadas com produtos de software. Exemplos de métricas de controle ou de processo são o esforço médio e o tempo necessário para reparar os defeitos reportados. Exemplos de métricas de predição são a complexidade de um módulo, a extensão média dos identificadores em um programa e o número de atributos e operações associados com os objetos em um projeto. Neste capítulo, pelo fato de me concentrar sobre as medições para prever a qualidade de produto de software, enfoco as métricas de predição. Explico as métricas de controle no Capítulo 28.

Geralmente, é impossível medir os atributos de qualidade de software diretamente. Os atributos de qualidade, como facilidade de manutenção, facilidade de compreensão e facilidade de uso são atributos externos que se relacionam a como os desenvolvedores e os usuários vêem o software. Eles são influenciados por muitos fatores e não existe uma maneira simples para medi-los. Em vez disso, você deve medir algum atributo interno do software (como seu tamanho) e presumir que há um relacionamento entre o que você pode medir e o que realmente quer conhecer. Idealmente, deve haver um relacionamento claro e validado entre os atributos de software internos e externos.

A Figura 27.6 mostra alguns atributos externos de qualidade que poderiam ser de interesse e atributos internos que poderiam estar relacionados com eles. O diagrama sugere que pode haver um relacionamento entre atributos internos e externos, mas não diz qual é o relacionamento. Se a medida do atributo interno deve ser uma previsão útil de uma característica externa de software, então três condições devem ser mantidas (Kitcheham, 1990):

1. O atributo interno deve ser medido com precisão.
2. Deve existir um relacionamento entre o que podemos medir e o atributo externo de comportamento no qual estamos interessados.
3. Esse relacionamento é compreendido, foi validado e pode ser expresso em termos de uma fórmula ou modelo.

Figura 27.5

Medições de predição e de controle.

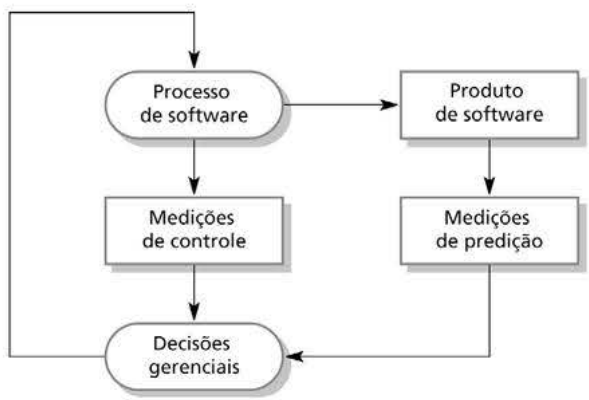
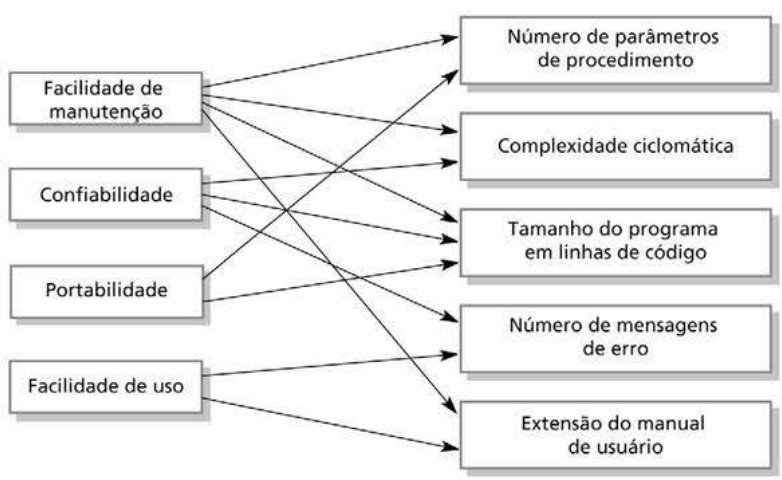


Figura 27.6

Relacionamentos entre atributos de software internos e externos.



A formulação de modelo envolve a identificação do formato funcional do modelo (linear, exponencial etc.) pela análise de dados coletados, a identificação de parâmetros que devem ser incluídos no modelo e o ajuste desses parâmetros usando dados existentes. Tal desenvolvimento de modelo deve ser confiável, requer experiência significativa em técnicas estatísticas e, de preferência, alguém com conhecimento e experiência em estatística deve definir o modelo a ser usado.

27.5.1 Processo de medição

Um processo de medição de software que pode ser parte de um processo de controle de qualidade é mostrado na Figura 27.7. Cada um dos componentes do sistema é analisado separadamente e os valores das métricas são comparados entre eles e, talvez, com dados históricos de medições coletados em projetos anteriores. As medições anômalas devem ser usadas para enfocar o esforço de garantia de qualidade sobre os componentes que podem ter problemas de qualidade.

Os principais estágios desse processo são:

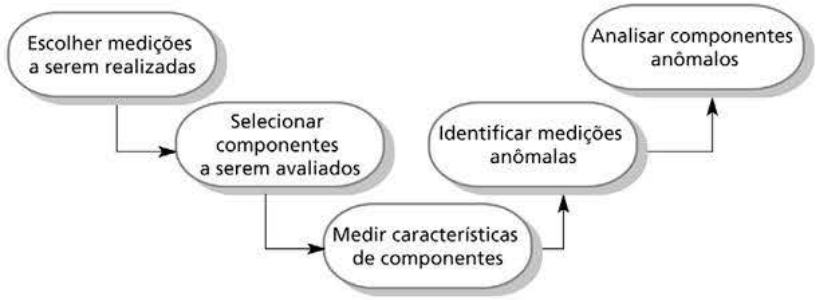
1. *Escolher medições a serem realizadas*. As questões que as medições se destinam a responder devem ser formuladas, e as medições necessárias para responder a essas questões devem ser definidas. As medições que não forem diretamente relevantes a essas questões não precisam ser coletadas. O paradigma GQM (*Goal-Question-Metric*) de Basili (Basili e Rombach, 1988), explicado no próximo capítulo, é uma boa abordagem a ser usada para decidir quais dados devem ser coletados.
2. *Selecionar componentes a serem avaliados*. Pode não ser necessário ou desejável avaliar valores de métricas para todos os componentes em um sistema de software. Em alguns casos, você pode selecionar um conjunto representativo de componentes para medição. Em outros, os componentes particularmente críticos, como os componentes centrais quase de uso constante, devem ser avaliados.
3. *Medir características de componentes*. Os componentes selecionados são medidos e os valores de métricas associadas são calculados. Isso normalmente envolve o processamento de representação de componentes (projeto, código etc.) usando uma ferramenta de coleta de dados automatizada. Essa ferramenta pode ser especialmente escrita ou pode já estar incorporada nas ferramentas CASE usadas em uma organização.
4. *Identificar medições anômalas*. Uma vez que as medições de componentes foram realizadas, você deve compará-las umas com as outras e com as medições anteriores que foram registradas em um banco de dados de medições. Você deve procurar por valores incomuns, altos ou baixos, para cada métrica, uma vez que isso sugere que poderia haver problemas com o componente que exibe esses valores.
5. *Analisar componentes anômalos*. Uma vez que os componentes que têm valores anômalos para métricas específicas foram identificados, você deve examinar esses componentes para decidir se os valores anômalos de métricas significam que a qualidade do componente está comprometida. Um valor anômalo de métrica para complexidade (digamos) não significa necessariamente um componente de baixa qualidade. Pode haver alguma outra razão para o alto valor e pode não significar que haja problemas de qualidade de componente.

Você deve sempre manter os dados coletados como um recurso da organização e manter registros históricos de todos os projetos, mesmo quando os dados não forem usados durante determinado projeto. Uma vez que um banco de dados de medições suficientemente grande foi estabelecido, você pode fazer comparações entre projetos e refinar as métricas específicas para ajudar a juntar as informações que a empresa necessita para o aprimoramento da qualidade.

27.5.2 Métricas de produto

As métricas de produto se dedicam às características do próprio software. Infelizmente, as características de software que podem ser facilmente medidas, como tamanho e complexidade ciclomática, não têm um relacionamento claro

Figura 27.7
Processo de medição de produto.



e consistente com atributos de qualidade, como facilidade de compreensão e facilidade de manutenção. Os relacionamentos variam dependendo dos processos e da tecnologia de desenvolvimento, e do tipo de sistema em desenvolvimento. Para descobrir e validar os relacionamentos entre os atributos externos e internos, você necessita coletar uma grande quantidade de dados de sistemas existentes. Você pode então usar esses dados para descobrir como os atributos de produto de software estão relacionados com as qualidades externas nas quais você está interessado.

As métricas de produtos se dividem em duas classes:

1. *Métricas dinâmicas* coletadas por meio de medições realizadas em um programa em execução.
2. *Métricas estáticas* coletadas por meio de medições realizadas em representações do sistema, como projeto, programa ou documentação.

Esses tipos de métricas estão relacionados com diferentes atributos de qualidade. As métricas dinâmicas ajudam a avaliar a eficiência e a confiabilidade de um programa. As métricas estáticas ajudam a avaliar a complexidade, a facilidade de compreensão e a facilidade de manutenção de um sistema de software.

As métricas dinâmicas estão usualmente relacionadas bem estreitamente com os atributos de qualidade de software. É bem fácil medir o tempo de execução necessário para funções específicas e para avaliar o tempo para iniciar um sistema. Essas métricas estão relacionadas diretamente com a eficiência do sistema. Similarmente, o número de falhas de sistema e o tipo de falha podem ser registrados e relacionados diretamente com a confiabilidade do software, conforme explicado no Capítulo 24.

As métricas estáticas, por outro lado, têm um relacionamento indireto com atributos de qualidade. Um grande número dessas métricas foi proposto e muitos experimentos tentaram derivar e validar os relacionamentos entre essas métricas com a complexidade, a facilidade de compreensão e a facilidade de manutenção do sistema. A Tabela 27.4 descreve várias métricas estáticas de produto que foram usadas para avaliação de qualidade. Além delas, a extensão de programa ou de componente e complexidade de controle parecem ser as previsões mais confiáveis de facilidade de compreensão, complexidade e facilidade de manutenção de sistema.

As métricas na Tabela 27.4 são genéricas, mas foram também propostas métricas orientadas a objetos mais específicas (Tabela 27.5). As métricas orientadas a objetos mais conhecidas foram propostas por Chidamber e Kemerer (Chidamber e Kemerer, 1994) e existem algumas ferramentas disponíveis para coletá-las. Algumas dessas métricas foram derivadas de métricas mais antigas mostradas na Tabela 27.4, mas outras são exclusivas para sistemas orientados a objetos. El-Aman (El-Aman, 2001), em excelente revisão sobre métricas orientadas a objetos, explica alguns desses estudos e conclui que ainda

Tabela 27.4 Métricas estáticas do produto de software

Métrica de software	Descrição
Fan-in/Fan-out	Fan-in é uma medida do número de funções ou métodos que chamam alguma outra função ou método (digamos X). Fan-out é o número de funções chamadas pela função X. Um valor alto para fan-in significa que X está firmemente acoplado com o resto do projeto, e mudanças em X terão grande impacto. Um alto valor para fan-out sugere que a complexidade geral de X pode ser alta devido à complexidade da lógica de controle necessária para coordenar os componentes chamados.
Extensão de código	É uma medida do tamanho de um programa. Geralmente, quanto maior for o tamanho do código de um componente, mais complexo e propenso a erros esse componente será. A extensão de código foi mostrada como uma das métricas mais confiáveis de previsão de propensão a erros em componentes.
Complexidade ciclomática	É uma medida da complexidade de controle de um programa. Essa complexidade de controle pode estar relacionada à facilidade de compreensão do programa. Explico como calcular a complexidade ciclomática no Capítulo 22.
Extensão de identificadores	É uma medida da extensão média de identificadores distintos em um programa. Quanto maiores forem os identificadores, mais eles serão significativos e, por isso, mais compreensível será o programa.
Profundidade de aninhamento de declarações condicionais	É uma medida da profundidade de aninhamento de declarações IF em um programa. As declarações IF profundamente aninhadas são difíceis de compreender e são potencialmente propensas a erros.
Índice de *Fog*	É uma medida da extensão média das palavras e das sentenças em documentos. Quanto maior for o valor para o índice de *Fog*, mais difícil será a compreensão de documentos.

Tabela 27.5 Métricas orientadas a objetos

Métrica orientada a objetos	Descrição
Profundidade de árvore de herança	Representa o número de níveis discretos na árvore de herança da qual as subclasses herdam atributos e operações (métodos) das superclasses. Quanto maior for a profundidade da árvore de herança, mais complexo será o projeto. Muitas classes de objeto podem ser compreendidas ao se compreender as classes de objeto das folhas da árvore.
Fan-in/fan-out de método	Está diretamente relacionado com fan-in e fan-out, conforme descrito na Tabela 27.4, e significa essencialmente a mesma coisa. Contudo, pode ser apropriado fazer uma distinção entre chamadas de outros métodos dentro do objeto e chamadas com base em métodos externos.
Métodos ponderados por classe	Este é o número de métodos incluídos em uma classe, ponderados pela complexidade de cada método. Portanto, um método simples pode ter uma complexidade de 1 e um método grande e complexo pode ter um valor muito mais alto. Quanto maior for o valor dessa métrica, mais complexa será a classe de objeto. Os objetos complexos são, provavelmente, mais difíceis de compreender. Eles podem não ser logicamente coesos, de modo que não podem ser reusados eficazmente como superclasses em uma árvore de herança.
Número de operações sobrescritas	Este é o número de operações, em uma superclasse, sobrescritas em uma subclasse. Um valor alto dessa métrica indica que a superclasse usada pode não ser uma classe-mãe apropriada para a subclasse.

não temos evidências suficientes para compreender como as métricas orientadas a objetos se relacionam com as qualidades externas de software.

As métricas específicas relevantes dependem do projeto, das metas da equipe de gerenciamento de qualidade e do tipo de sistema em desenvolvimento. Todas essas métricas mostradas nas tabelas 27.4 e 27.5 podem ser úteis em algumas situações. Igualmente, contudo, haverá outras situações em que elas serão inadequadas. Quando se introduz a medição de software como parte do processo de gerenciamento de qualidade, as organizações devem experimentar para descobrir as métricas mais apropriadas para as suas necessidades.

27.5.3 Análise de medições

Um dos problemas com a coleta de dados quantitativos sobre o software e o projeto de software é a compreensão do que aqueles dados significam. É fácil interpretar errado os dados e fazer inferências incorretas. As medições devem ser cuidadosamente analisadas para compreender o que elas realmente significam.

Para ilustrar como os dados coletados podem ser interpretados de diferentes maneiras, considere o seguinte cenário. Para tornar isso mais fácil de compreender, usei um processo em vez de uma métrica de produto, mas a mensagem essencial é a mesma; as razões para a mudança podem, muitas vezes, ser difíceis de compreender.

> Um gerente decide monitorar o número de solicitações de mudança submetidas pelos clientes baseado em uma suposição de que existe um relacionamento entre essas solicitações de mudança e a facilidade de uso e a adequação de produto. Quanto maior o número de solicitações de mudança, menos o software atende às necessidades do cliente.
>
> O processamento de solicitações de mudança e a mudança de software são dispendiosos. A organização, portanto, decide modificar seu processo para aumentar a satisfação do cliente e, ao mesmo tempo, reduzir os custos de mudança. A intenção é que as mudanças de processo resultarão em produtos melhores e menos solicitações de mudança.
>
> As mudanças de processo são iniciadas para aumentar o envolvimento do cliente no processo de projeto de software. Testes beta de todos os produtos são introduzidos e as modificações solicitadas pelo cliente são incorporadas ao produto entregue. Novas versões do produto, desenvolvidas com esse processo modificado, são entregues. Em alguns casos, o número de solicitações de mudança é reduzido; em outros, aumenta. O gerente está confuso e não pode avaliar os efeitos das mudanças de processo sobre a qualidade de produto.

Para entender por que isso pode ocorrer, você deve entender por que as solicitações de mudança são feitas. Uma das razões é que o software entregue não faz o que os clientes gostariam que ele fizesse. Outra possibilidade é que o software é muito bom, ampla e intensamente usado e, algumas vezes, para finalidades para as quais ele não foi originalmente projetado. Pelo fato de haver muitas pessoas que estão usando o software, é natural que mais solicitações de mudança sejam geradas.

Uma terceira possibilidade é que a empresa que produz o software esteja suscetível às solicitações de mudança do cliente. Os clientes estão, portanto, satisfeitos com os serviços que recebem. Eles geram grande quantidade de solicitações de mudança porque sabem que essas solicitações serão tratadas seriamente. Suas sugestões serão provavelmente incorporadas nas versões posteriores do software.

O número de solicitações de mudança poderia diminuir porque as mudanças de processo foram efetivadas e tornaram o software mais usável e adequado. Alternativamente, o número poderia diminuir porque o produto perdeu participação no mercado para um produto rival. Existem, conseqüentemente, poucos usuários do produto. O número de solicitações de mudança poderia aumentar porque existem mais usuários, porque o processo de teste beta convenceu os usuários de que a empresa está disposta a fazer mudanças ou porque as configurações de testes beta não eram típicas para o maior uso do programa.

Para analisar os dados de solicitações de mudança, você não necessita simplesmente saber o número de solicitações de mudança. Você necessita saber quem fez a solicitação, como eles usam o software e por que a solicitação foi feita. Você necessita saber se existem fatores externos, como modificações no procedimento de solicitações de mudança, ou mudanças de mercado que poderiam ter algum efeito. Com essas informações, é possível então identificar se as mudanças de processo foram eficazes no aumento da qualidade de produto.

Isso mostra que a interpretação de dados quantitativos sobre um produto ou um processo é subjetiva. Os processos e os produtos em medição não são isolados de seu ambiente e as mudanças para esse ambiente podem tornar inválidas as comparações de dados. Os dados quantitativos sobre atividades humanas não podem sempre ser considerados pelo valor em si. As razões implícitas que poderiam ser consideradas para os valores medidos devem ser investigadas.

PONTOS-CHAVE

- O gerenciamento de qualidade de software dedica-se a assegurar que o software tem baixo número de defeitos e que atinge os padrões necessários de facilidade de manutenção, confiabilidade, portabilidade etc. As atividades de gerenciamento de qualidade incluem a garantia de qualidade que estabelece padrões para o desenvolvimento de software, o planejamento de qualidade e o controle de qualidade que verifica o software em relação aos padrões definidos.
- Você deve documentar um conjunto de procedimentos de garantia de qualidade em um manual de qualidade da organização. Ele pode basear-se no modelo genérico de um manual de qualidade nos padrões ISO 9000.
- Os padrões de software são importantes para a garantia de qualidade quando representam uma identificação da 'melhor prática'. O processo de controle de qualidade dedica-se a verificar se o processo de software e o software em desenvolvimento estão em conformidade com esses padrões.
- As revisões de produtos relacionados ao processo de software envolvem uma equipe de pessoas que verificam se os padrões de qualidade estão sendo seguidos. As revisões são a técnica mais amplamente usada para a avaliação de qualidade.
- As medições de software podem ser usadas para juntar dados quantitativos sobre o software e o processo de software. Você pode ser capaz de usar os valores das métricas de software coletados para fazer inferências sobre a qualidade de produto e de processo.
- As métricas de qualidade de produto são particularmente valiosas para destacar componentes anômalos que podem ter problemas de qualidade. Esses componentes devem então ser analisados mais detalhadamente.
- Não existem métricas de software padronizadas e universalmente aplicáveis. As organizações devem selecionar suas métricas e analisar as medições baseadas no conhecimento e nas circunstâncias locais.

LEITURAS SUGERIDAS

Software quality assurance: from theory to implementation. Excelente e atualizada visão sobre os princípios e práticas de garantia de qualidade de software. Inclui uma explicação de padrões, como ISO 9001 (D. Galin, 2004, Addison-Wesley.)

Metrics and models for software quality engineering, 2nd ed. Essa é uma explicação muito abrangente sobre métricas de software que aborda métricas de processo, de produto e orientadas a objetos. O livro trata também da matemática necessária para desenvolver e compreender os modelos baseados em medições de software (S. H. Kan, 2003, Addison-Wesley.)

"Making sense of measurement for small organisations". Este é um artigo interessante sobre a aplicação prática de métricas. Destaca que todos os usos de métricas devem levar em conta o contexto (K. Kautz, *IEEE software,* março/abril de 1999).

A quantitative approach to software management: the ami handbook. É um excelente guia do tipo 'como fazer' que apresenta um programa de medições e o uso de resultados para o aprimoramento do processo. Infelizmente, está esgotado, mas deve estar disponível em boas livrarias (K. Pulford, et al., 1996, Addison-Wesley.)

EXERCÍCIOS

27.1 Explique por que um processo de software de alta qualidade deve conduzir a produtos de software de alta qualidade. Explique os possíveis problemas com esse sistema de gerenciamento de qualidade.

27.2 Explique como os padrões podem ser usados para captar o conhecimento da organização sobre métodos eficazes de desenvolvimento de software. Sugira quatro tipos de conhecimentos que podem ser captados em padrões organizacionais.

27.3 Explique a avaliação de qualidade de software de acordo com os atributos de qualidade mostrados no Quadro 27.1. Você deve considerar cada atributo por vez e explicar como ele pode ser avaliado.

27.4 Projete um formulário eletrônico que possa ser usado para registrar comentários de revisões e ser usado para remeter eletronicamente os comentários aos revisores.

27.5 Descreva, brevemente, os possíveis padrões que poderiam ser usados para:

- O uso de construções de controle em C, C++ ou Java.
- Relatórios que podem ser enviados para um projeto de formatura em uma universidade.
- O processo de fazer e aprovar mudanças em um programa (veja o Capítulo 29).
- O processo de comprar e instalar um novo computador.

27.6 Suponha que você trabalhe para uma organização que desenvolve produtos de banco de dados para sistemas de microcomputadores. Essa organização está interessada em quantificar seu desenvolvimento de software. Redija um relatório que sugira métricas apropriadas e aponte como elas podem ser coletadas.

27.7 Explique por que as métricas de projeto são, por si sós, um método inadequado para prever a qualidade do projeto.

27.8 Consulte a literatura e descubra outras métricas de projeto sugeridas, além das explicadas aqui. Considere essas métricas detalhadamente e avalie se elas podem ser de valor real.

27.9 Explique por que é difícil validar o relacionamento entre os atributos internos de produto, como a complexidade ciclomática e atributos externos, como a facilidade de manutenção.

27.10 Sugira como as medições de software automatizadas podem ser usadas para apoiar a *extreme programming* (explicada no Capítulo 17).

27.11 Os padrões de software reprimem a inovação tecnológica?

27.12 Um colega, que é um programador muito bom, produz software com baixo número de defeitos, mas ele ignora consistentemente os padrões de qualidade da organização. Como os gerentes da organização devem reagir em relação a esse comportamento?

28

Aprimoramento de processo

Objetivos

O objetivo deste capítulo é explicar como um processo de software pode ser aprimorado para produzir um software melhor. Após ler este capítulo, você:

- compreenderá os princípios do processo de aprimoramento de software e por que ele vale a pena;
- compreenderá como os fatores de processo de software influenciam a qualidade do software e a produtividade dos desenvolvedores;
- será capaz de desenvolver modelos simples de processos de software;
- compreenderá as noções de capacidade e maturidade de processo e a forma geral do modelo CMMI para o aprimoramento de processos.

Conteúdo

28.1 Qualidade de processo e de produto
28.2 Classificação de processos
28.3 Medição de processos
28.4 Análise e modelagem de processos
28.5 Mudança de processos
28.6 Framework CMMI de aprimoramento de processos

Conforme expliquei no Capítulo 27, há uma relação estreita entre a qualidade de um processo de desenvolvimento e a qualidade de produtos desenvolvidos por meio desse processo. Devido a esse fato, muitas empresas de engenharia de software têm se voltado para o aprimoramento de processos de software como maneira de aumentar a qualidade de seus softwares. *Aprimoramento de processo* significa compreender os processos existentes e alterá-los para incrementar a qualidade do produto e/ou reduzir custos e tempo de desenvolvimento. A maioria das literaturas referentes ao aprimoramento de processos enfoca o aperfeiçoamento do processo para aumentar a qualidade e, em particular, para reduzir o número de defeitos no software entregue. Quando isso for alcançado, os principais objetivos podem ser a redução dos custos ou dos prazos.

Os processos de softwares são intrinsecamente complexos e envolvem um grande número de atividades. Como os produtos, os processos também possuem atributos ou características, conforme mostrado na Tabela 28.1. Não é possível estabelecer o aprimoramento de processos que otimizem todos esses atributos simultaneamente. Por exemplo, se o seu objetivo é ter um processo de desenvolvimento rápido, então você deve reduzir a visibilidade do processo. Tornar um processo visível significa produzir documentos em intervalos regulares. Isso, inevitavelmente, atravanca o andamento do processo.

Tabela 28.1 Características de processo

Característica de processo	Descrição
Facilidade de compreensão	Em que extensão o processo está explicitamente definido e com que facilidade se compreende a definição de processo?
Visibilidade	As atividades do processo culminam em resultados claros de maneira que o progresso do processo é visível externamente?
Facilidade de apoio	Em que extensão as ferramentas CASE podem ser usadas para apoiar as atividades de processo?
Aceitabilidade	O processo definido é aceitável e usável pelos engenheiros responsáveis pelos produtos de software?
Confiabilidade	O processo está projetado de maneira que seus erros são evitados ou percebidos antes que resultem em erros de produto?
Robustez	O processo pode continuar apesar do surgimento de problemas inesperados?
Facilidade de manutenção	O processo pode evoluir para refletir requisitos de mudanças organizacionais ou aprimoramentos de processo identificados?
Rapidez	Com que rapidez o processo de entrega de um sistema pode ser concluído com base em determinada especificação?

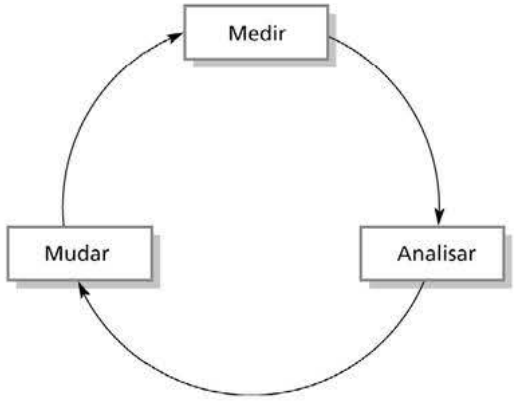
O aprimoramento de processos não significa, simplesmente, adotar métodos específicos ou ferramentas, ou algum modelo de processo que tenha sido empregado em outro lugar. Embora as organizações que desenvolvem o mesmo tipo de software, claramente, tenham muita coisa em comum, há sempre fatores organizacionais locais, procedimentos e padrões que influenciam o processo. Você raramente será bem-sucedido na introdução do aprimoramento de processos se simplesmente tentar mudar o processo para um que é empregado em algum outro lugar. Você deve sempre olhar o aprimoramento de processos como algo específico para uma organização ou parte de uma organização maior.

O aprimoramento de processos é uma atividade cíclica, conforme mostrado na Figura 28.1. Ela compreende três estágios principais:

1. *Medição de processo.* São medidos os atributos do projeto ou do produto atual. O objetivo é aprimorar as medidas de acordo com os objetivos da organização envolvida no aprimoramento de processos.
2. *Análise de processo.* O projeto atual é avaliado e os seus pontos fracos bem como respectivos gargalos são identificados. Os modelos de processo que descrevem o processo são normalmente desenvolvidos nesse estágio.
3. *Mudança de processo.* As mudanças para o processo que foram identificadas durante a análise são introduzidas.

Abordo cada uma dessas atividades separadamente nas seções seguintes deste capítulo. Cada estágio do processo dura muitos meses; o aprimoramento de processos é uma atividade de longo prazo. É também uma atividade contínua uma vez que, quaisquer que sejam os novos processos introduzidos, o ambiente de negócios mudará e esses processos, em si, terão de envolver e levar em conta essas mudanças.

Figura 28.1
Ciclo de aprimoramento do processo.



28.1 Qualidade de processo e de produto

O aprimoramento de processos baseia-se na hipótese de que a qualidade do processo de desenvolvimento de produto é crítica para a qualidade do produto. A noção de aprimoramento de processos é a idéia original do engenheiro americano W. E. Deming, que trabalhou com a indústria japonesa depois da Segunda Guerra Mundial para aumentar a qualidade. A indústria japonesa se comprometeu com o aprimoramento de processos durante muitos anos, o que a conduziu para o reconhecimento de alta qualidade dos produtos manufaturados japoneses.

Deming (e outros) introduziu o conceito do controle estatístico de qualidade. O conceito baseia-se em medições do número de defeitos de produtos e os relaciona com o processo. O objetivo é reduzir o número de defeitos de produtos pelo aprimoramento de processos até que o número possa ser repetível. Ou seja, os resultados de processo são previsíveis e o número de defeitos é reduzido. O processo é, então, padronizado e o ciclo de aprimoramentos futuros é iniciado.

Humphrey, no seu livro inicial sobre gerenciamento de processos (Humphrey, 1988), argumenta que as mesmas técnicas podem ser aplicadas na engenharia de software. Ele declara:

> *W. E. Deming, em seu trabalho com as indústrias japonesas após a Segunda Guerra Mundial, aplicou os conceitos de controle estatístico de processos na indústria. Embora haja diferenças importantes, esses conceitos são tão aplicáveis ao software quanto aos automóveis, câmeras, relógios de pulso e aço.*

Embora haja similaridades claras, não concordo com Humphrey a respeito de que os resultados da engenharia da manufatura possam ser diretamente transferidos para a engenharia de software. Quando a manufatura é envolvida, o relacionamento processo/produto é muito óbvio. Ao aprimorar um processo para evitar os defeitos, isso levará a produtos mais aprimorados. Essa ligação é menos óbvia quando o produto é intangível e dependente, em alguma extensão, de processos intelectuais que não podem ser automatizados. A qualidade de software não depende de um processo de manufatura, mas de um processo de projeto no qual as capacidades individuais humanas são significativas. Em algumas classes de produtos, o processo usado é o mais significante e determinante para a qualidade de produto. Entretanto, para aplicações inovadoras em particular, as pessoas envolvidas no processo são mais importantes do que o processo em si.

Para produtos de software ou para quaisquer outros produtos intelectuais como livros ou filmes, em que a qualidade do produto depende de seu projeto, quatro fatores principais afetam essa qualidade. Esses fatores são mostrados na Figura 28.2.

A influência de cada um dos fatores depende do tamanho e do tipo de projeto. Para sistemas de grande porte que incluem subsistemas separados, desenvolvidos por equipes que podem trabalhar em diferentes localidades, o principal fator que afeta a qualidade de produto é o processo de software. Os principais problemas associados a grandes projetos são integração, gerenciamento de projeto e comunicação. Há geralmente uma combinação de habilidades e de experiência nos membros de equipe e, devido ao processo de desenvolvimento ocorrer normalmente ao longo de muitos anos, a equipe de desenvolvimento é volátil. Isso pode mudar completamente durante o tempo de vida do projeto. Portanto, particularmente, pessoas bem talentosas e experientes não têm efeito dominante sobre o tempo de vida do projeto.

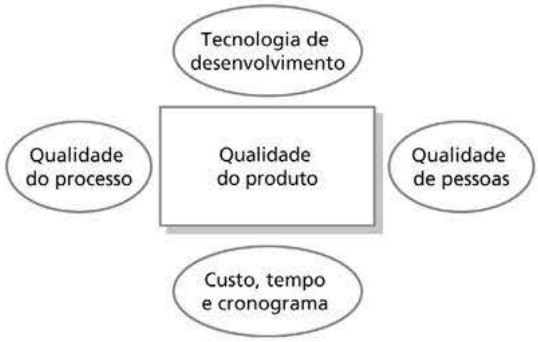
No caso de projetos de pequeno porte, entretanto, nos quais há uma quantidade reduzida de membros na equipe de projeto, a qualidade da equipe de desenvolvimento é mais importante do que o processo usado. Se a equipe tem alto nível de habilidade e experiência, a qualidade do produto é provavelmente alta. Se a equipe for inexperiente e despreparada, um bom processo pode limitar os danos, mas não proporcionará uma alta qualidade de software simplesmente pela sua aplicação.

Quando as equipes de projeto são pequenas, é essencialmente importante o uso de uma boa tecnologia de desenvolvimento. Uma equipe pequena não pode dedicar muito tempo aos procedimentos administrativos tediosos. Os membros da equipe dedicam grande parte do seu tempo em atividades de projeto e de programação de sistemas, de modo que boas ferramentas podem afetar significativamente a sua produtividade. Em projetos de grande porte, um nível básico de tecnologia de desenvolvimento é essencial para o gerenciamento de informações. Paradoxalmente, ferramentas CASE sofisticadas são menos importantes em projetos de grande porte. Os membros da equipe gastam uma pequena parcela do seu tempo em atividades de desenvolvimento e uma grande parte do tempo em comunicação e no entendimento de outras partes do sistema. Esse é o fator dominante que afeta a sua produtividade. Nesse caso, as ferramentas de desenvolvimento não fazem nenhuma diferença.

A base do retângulo mostrada na Figura 28.2 é realmente crítica. Se um projeto, independentemente do seu tamanho, estiver abaixo do orçado ou for planejado com uma programação irreal de entrega, a qualidade do produto será afetada. Um bom processo requer recursos para sua efetiva implementação. Se esses recursos forem insuficientes, o processo não será eficiente. Se, por outro lado, os recursos forem inadequados, somente profissionais excepcionais poderão salvar um projeto. Do mesmo modo, se o déficit for muito grande, a qualidade do produto será degradada.

Em todos os casos, a causa real dos problemas de qualidade de software não é devida ao fraco gerenciamento, a um processo inadequado ou um treinamento fraco sobre qualidade. Ao contrário, é o fato de que as organizações devem competir para sobreviver. Para obter um contrato, uma empresa pode subestimar o esforço necessário ou prometer entrega rápida de um sistema. Na tentativa de atender a esses compromissos, as empresas podem sacrificar a qualidade do software.

Figura 28.2
Fatores principais de qualidade de produtos de software.



28.2 Classificação de processos

Processos de software podem ser observados em todas as organizações, desde empresas de uma só pessoa até grandes multinacionais. Esses processos são de diferentes tipos que dependem do grau de formalidade dos processos, dos tipos de produtos desenvolvido, do tamanho da organização etc. Existem quatro classes de processos de software:

1. *Processos informais*. Quando não há um modelo de processo rigorosamente definido, a equipe de desenvolvimento escolhe o processo que será usado. Processos informais podem usar procedimentos formais como gerenciamento de configurações, mas os procedimentos e os relacionamentos entre os procedimentos são definidos quando forem exigidos pela equipe de desenvolvimento.
2. *Processos gerenciados*. Um modelo de processo definido é usado para conduzir o processo de desenvolvimento. O modelo de processo define os procedimentos, sua programação e os relacionamentos entre os procedimentos.
3. *Processos metódicos*. Quando algum método, ou métodos, definido de desenvolvimento (como métodos sistemáticos para projeto orientado a objetos) é usado, esses processos se beneficiam do apoio de ferramenta CASE para processos de análise e de projeto.
4. *Processos de aprimoramento*. São processos que têm inerentemente objetivos de aprimoramento e têm um orçamento específico para os aprimoramentos e procedimentos a fim de introduzir esses aprimoramentos. Como parte disso, uma medição quantitativa de processo pode ser introduzida.

Essas classificações obviamente se sobrepõem e um processo pode ser enquadrado em diferentes classes. Por exemplo, o processo pode ser informal tal como ele foi escolhido pela equipe de desenvolvimento. A equipe pode optar pelo uso de um método específico de desenvolvimento. Eles também devem ter capacidade de aprimoramento de processos. Nesse caso, o processo seria classificado como *informal, metódico* e *de aprimoramento.*

Essas classificações são úteis porque servem como base para o aprimoramento de processos multidimensionais. As classificações auxiliam as organizações a escolher um processo apropriado aos requisitos específicos de desenvolvimento do produto. A Figura 28.3 mostra os diferentes tipos de produtos e de processos que poderiam ser usados nos desenvolvimento. Processos de aprimoramento não são mostrados nesse diagrama porque qualquer tipo de processo pode ser um processo de aprimoramento.

As classes de sistema mostradas do lado direito da Figura 28.3 podem se sobrepor. No entanto, sistemas pequenos que foram submetidos a uma reengenharia podem ser desenvolvidos por meio de um processo metódico. Sistemas grandes necessitam sempre de um processo gerenciado. Entretanto, os métodos de projeto não são adequados a todos os tipos de aplicações, e sistemas grandes em particular podem incluir subsistemas de diferentes tipos. Em vez de impor um único método aos projetistas, esses sistemas podem ser desenvolvidos com a utilização de um processo gerenciado que não é baseado em nenhum método específico.

A classificação de processo provê a base para a escolha de um processo certo para um produto em desenvolvimento. Por exemplo, um programa que apoiará uma transição de um tipo de sistema computacional para outro tem ciclo de vida relativamente curto. Seu desenvolvimento não exige padronizações ou procedimentos de gerenciamento apropriados para o software que será usado por muitos anos.

A classificação de processo reconhece que o processo afeta a qualidade de produto. Não considera, no entanto, que o processo é sempre o fator dominante. Ela fornece uma base para o aprimoramento de diversos tipos de processos. Diferentes tipos de aprimoramento de processos podem ser aplicados a diferentes tipos de processos. Por exemplo, os aprimoramentos de processos metódicos podem ser baseados em um método melhor de treinamento, melhor integração de requisitos e de projeto, ferramentas CASE aprimoradas etc.

Figura 28.3
Aplicabilidade do processo.

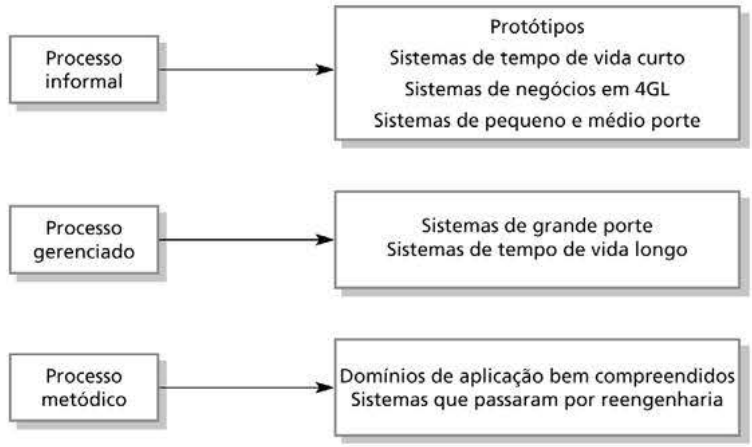
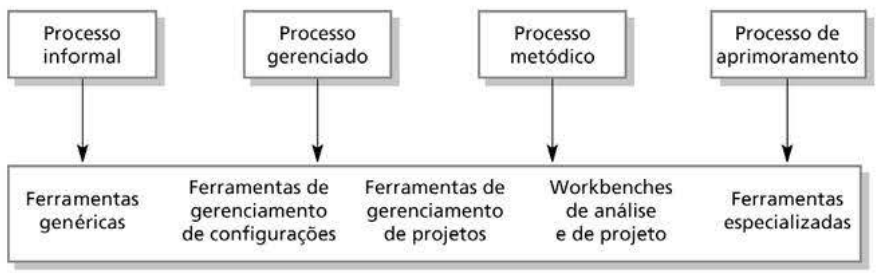


Figura 28.4
Apoio de ferramentas de processo.



A maioria dos processos de software possui certo apoio de ferramentas CASE atualmente; são os *processos apoiados*. Os workbenches de análise e de projeto são usados para apoiar os processos metódicos. Entretanto, os processos podem ter outros tipos de ferramentas de apoio (por exemplo, ferramentas de prototipação, ferramentas de testes) independentemente se é usado um método estruturado de projeto.

As ferramentas de apoio que podem ser eficientes no apoio aos processos dependem da classificação de processos. Por exemplo, ferramentas genéricas, como linguagens de prototipação, compiladores, debuggers e processadores de texto, podem ser usadas nos processos informais. No entanto, processos informais raramente usam ferramentas mais especializadas de maneira consistente. A Figura 28.4 mostra que um espectro de ferramentas pode ser usado no desenvolvimento de software. A eficiência de ferramentas específicas depende do tipo de processo usado.

Ferramentas CASE para análise e para projeto são provavelmente mais adequadas em custo em um processo metódico. Ferramentas especializadas provêem apoio específico às atividades individuais de processo. Por exemplo, uma equipe envolvida em desenvolvimento em múltiplas localidades pode criar uma ferramenta de display que mostra o andamento do trabalho em cada localidade. Algumas vezes, essas ferramentas especializadas podem ser desenvolvidas especialmente para aprimorar o processo.

28.3 Medição de processos

Medições de processo são dados quantitativos sobre o processo de software. Humphrey (Humphrey, 1989), em seu livro sobre aprimoramento de processos, argumenta que a medição de atributos de processo e de produto é essencial para o aprimoramento de processos. Ele também sugere que a medição tenha importante papel a desempenhar no aprimoramento de processos pequenos e pessoais (Humphrey, 1995). Medições de processo podem ser usadas para avaliar se a eficiência de um processo foi aprimorada. Por exemplo, o esforço e o tempo dedicados às atividades de testes podem ser monitorados. Aprimoramentos eficientes no processo de teste devem reduzir o esforço, o tempo de teste, ou ambos. No entanto, as próprias medições de processos não podem ser usadas para determinar se a qualidade de um produto foi aumentada. Os dados de qualidade de produto (veja o Capítulo 27) devem ser também coletados e relacionados às atividades de processo.

Três classes de métrica de processo podem ser coletadas:

1. *O tempo para que um processo específico possa ser concluído*. Esse pode ser o tempo total dedicado ao processo, tempo de calendário, tempo despendido no processo por determinados engenheiros etc.

2. *Os recursos necessários para um processo específico.* Os recursos poderiam incluir o esforço total em termos de pessoas-dia, custos de viagem e recursos computacionais.
3. *O número de ocorrências de um evento específico.* Exemplos de eventos que poderiam ser monitorados incluem o número de defeitos descobertos durante uma inspeção de código, o número de mudanças de requisitos solicitadas e o número médio de linhas de código modificadas em resposta a uma mudança de requisitos.

Os primeiros dois tipos de medições podem ser usados para descobrir se as mudanças de processo aumentaram sua eficiência. Digamos que existam três pontos fixos no processo de desenvolvimento de software como aceitação de requisitos, a conclusão do projeto de arquitetura e da geração de dados de teste. Você pode ser capaz de medir o tempo e o esforço necessários para mover-se de um desses pontos fixos para outro. Após as mudanças serem introduzidas, as medições de atributos de sistema podem mostrar se as mudanças de processos beneficiaram realmente a redução de tempo ou de esforço necessário.

As medições do número de eventos que ocorrem podem ter uma relação mais direta com a qualidade de software. Por exemplo, o incremento do número de defeitos descobertos pela mudança do processo de inspeção de programa poderá, provavelmente, refletir no aumento da qualidade do produto. Entretanto, esse fato deve ser confirmado pelas medições subseqüentes de produtos.

A dificuldade fundamental da medição de processo é justamente saber o que medir. Basili e Rombach (Basili e Rombach, 1988) propuseram o que eles chamam de paradigma Objetivo-Questão-Métrica (GQM — *Goal-Question-Metric*). Ele é usado para auxiliar a decidir quais medições devem ser realizadas e como elas devem ser usadas. Essa abordagem conta com a identificação de:

1. *Objetivos.* Definem o que a organização está tentando obter. Exemplos de objetivos podem ser o aumento da produtividade de programadores, redução do tempo de desenvolvimento de produto e incremento da confiabilidade de produto.
2. *Questões.* São refinamentos dos objetivos, em que áreas específicas de incertezas relacionadas aos objetivos são identificadas. Normalmente, um objetivo terá um número de questões associadas que necessitam ser respondidas. Exemplos de questões relacionadas aos objetivos são:
   - Como o número de linhas de código depuradas pode ser aumentado?
   - Como o tempo necessário para finalizar os requisitos de produto pode ser reduzido?
   - Como avaliações mais confiáveis e eficientes podem ser feitas?
3. *Métricas.* São as medições que necessitam ser coletadas para auxiliar a responder às questões e confirmar se os aprimoramentos de processos alcançaram o objetivo desejado. Nos exemplos apresentados anteriormente, as medições a serem realizadas incluem a produtividade de programadores individuais em linhas de código e do seu nível de experiência, o número de comunicações formais entre o cliente e o contratado para cada mudança de requisitos e o número de testes necessários para causar uma falha de produto.

A vantagem dessa abordagem aplicada ao aprimoramento de processos é que ela separa assuntos organizacionais (os objetivos) de processo específico (as questões). Enfoca a coleta de dados e sugere que os dados coletados devam ser analisados em diferentes maneiras, dependendo da questão que se deseja responder. Basili e Green (Basili e Green, 1993) descrevem como esse paradigma pode ser usado em um programa de aprimoramento de processos baseados em medições por muito tempo.

A abordagem GQM foi desenvolvida e combinada com o modelo de maturidade de capacitação do SEI (Paulk, et al. 1995) no método ami (Pulford, et al. 1996) de aprimoramento de processos de software. Os desenvolvedores do método ami propuseram uma abordagem por estágios de aprimoramento de processos nos quais a medição é introduzida quando uma organização integrou alguma disciplina em seus processos. O método ami fornece um guia e sugestões práticas para a implementação de aprimoramento de processos baseada em medições.

28.4 Análise e modelagem de processos

Análise e modelagem de processo envolvem o estudo dos processos existentes e o desenvolvimento de um modelo abstrato desses processos que apreendam as características principais. Esses modelos ajudam você a compreender o processo e comunicar essa compreensão a outros. Em todo o livro, usei fragmentos de modelo de processo para explicar atividades específicas como engenharia de requisitos e projeto de software. Sugeri na Figura 28.1 que a análise de processo siga a medição de processo. Isso é uma simplificação porque, na realidade, essas atividades são entrelaçadas. Você precisa

realizar a análise para saber o que medir e, ao realizar as medições, você inevitavelmente desenvolverá um profundo conhecimento do processo que está sendo medido.

A análise de processo está relacionada com o estudo de processos existentes no sentido de compreender os relacionamentos entre as partes do processo. Os estágios iniciais de análise de processo são qualitativos: o analista está simplesmente tentando descobrir as características principais do modelo. Os estágios posteriores podem ser quantitativos e fazem uso das medições de processo coletadas. Após completar a análise, você deve descrever e documentar o processo usando um modelo de processo de software (Huff, 1996).

O ponto inicial para a análise de processo deve ser qualquer modelo formal de processo usado. Muitas organizações possuem um modelo formal que pode ser imposto a todos pelo cliente de software. Esse padrão define as atividades críticas e entregas no ciclo de vida que devem ser produzidas.

Modelos formais podem servir como ponto de partida útil para a análise de processo. No entanto, raramente incluem detalhes suficientes ou refletem as atividades reais do desenvolvimento de software. Modelos formais de processos são suficientemente abstratos e definem somente as atividades principais de processos e de entregas. É em geral necessário olhar mais internamente o modelo para encontrar o processo real. Além disso, o processo real muitas vezes difere significativamente do modelo formal; no entanto, as entregas especificadas serão geralmente produzidas.

As técnicas de análise de processo incluem:

1. *Questionários e entrevistas*. Os engenheiros que trabalham em um projeto são questionados sobre como o projeto está realmente caminhando. As respostas a um questionário formal são refinadas durante as entrevistas pessoais com os envolvidos no processo. A discussão pode ser estruturada com base em uma versão do modelo de processo refinado conforme novas informações tornam-se disponíveis.
2. *Estudos etnográficos.* Como explicado no Capítulo 7, estudos etnográficos podem ser usados para compreender a natureza do desenvolvimento do software como atividade humana. Tal análise revela as sutilezas e as complexidades que não podem ser descobertas com o uso de outras técnicas.

Cada uma dessas abordagens tem vantagens e desvantagens. A análise baseada em questionários pode ser raramente realizada com rapidez uma vez que questões certas têm de ser identificadas. No entanto, se as questões forem mal formuladas ou inadequadas, você terminará com um modelo incompleto ou impreciso do processo. Além disso, a análise baseada em questionários pode parecer uma avaliação. Os engenheiros questionados podem dar respostas que eles pensam que você quer ouvir em vez de falar a verdade sobre o processo usado.

As entrevistas com pessoas envolvidas no processo são mais informais do que os questionários. Você inicia com um roteiro preparado de questões, mas o adapta conforme as respostas que estão sendo obtidas de pessoas diferentes. Se você der aos participantes uma oportunidade para discutir assuntos de maneira mais ampla, pode descobrir que os participantes falam sobre os problemas de processo, as maneiras como o processo é modificado na prática etc.

É mais provável que a análise etnográfica descubra o verdadeiro processo em utilização. Entretanto, é uma atividade prolongada, que poderá durar muitos meses. Ela conta com uma observação externa de processo com base no instante em que ele foi aprovado. Para realizar uma análise completa, você precisa se envolver desde os estágios iniciais de um projeto até a entrega e a manutenção do produto. Para projetos grandes, que poderiam durar muito anos, é claro que isso é impraticável em tais circunstâncias. A análise etnográfica, portanto, é mais útil quando é necessário um conhecimento profundo de fragmentos do processo. Você realiza um estudo em pequena escala enfocando os detalhes do processo.

Modelos de processo são visões simplificadas do processo de software que mostram as atividades e as saídas dos processos. Os modelos de processos usados neste livro apresentam uma visão de muito alto nível dos processos relacionados. Nesse nível de abstração, esses processos são os mesmos em muitas organizações. Contudo, os modelos genéricos têm diferentes instâncias em empresas diferentes dependendo do tipo de software em desenvolvimento e do ambiente organizacional. Os modelos de processo detalhados não são normalmente os mesmos entre as organizações.

Modelos de processo abstratos ou genéricos são bases úteis para discussão de processos. Entretanto, não possuem informações suficientes para análise e aprimoramento de processos. O aprimoramento de processos necessita de informações sobre atividades, entregas, pessoas, comunicações, cronogramas e outros processos organizacionais que afetam o processo de desenvolvimento de software. A Tabela 28.2 explica o que poderia ser incluído em um modelo de processo detalhado.

Você deve registrar o *timing* e as dependências entre atividades, entregas e comunicações no modelo de processo. Algumas vezes, as atividades podem ser realizadas em paralelo e em outras ocasiões elas devem ocorrer em seqüência. Podem ser intercaladas de modo que uma mesma pessoa é envolvida em diversas atividades. As entregas podem depender de outras entregas ou de comunicações entre engenheiros que trabalham no processo.

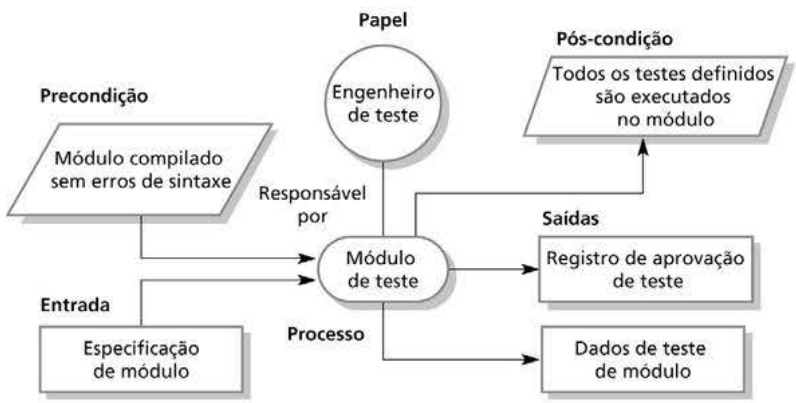
Nos exemplos de modelos de processo neste livro, mostro uma seqüência aproximada de atividades da esquerda para a direita. As atividades que podem ser realizadas em paralelo estão, tanto quanto possível, alinhadas verticalmente.

Os modelos de processo detalhados são extremamente complexos. Por exemplo, considere os fragmentos de processo mostrados nas figuras 28.5 e 28.6. Eles descrevem o processo de teste de um único módulo de um grande sistema que usa

Tabela 28.2 Elementos de um modelo de processo

Elemento de modelo de processo	Descrição
Atividade (mostrada como um retângulo de bordas arredondadas sem sombreado)	Uma atividade tem objetivo e condições de entrada e saída claramente definidos. Exemplos de atividades são a preparação de um conjunto de dados de teste de um módulo, codificação de uma função ou um módulo e a prova de leitura de um documento. Geralmente, uma atividade é atômica, ou seja, é de responsabilidade de uma pessoa ou de um grupo. Não é decomposta em subatividades.
Processo (mostrado como um retângulo de bordas arredondadas com sombreado)	Processo é um conjunto coerente de atividades cujo objetivo está geralmente de acordo com uma organização. Exemplos de processos são a análise de requisitos, o projeto de arquitetura e o planejamento de testes.
Entrega (mostrada como um retângulo com sombreado)	Entrega é uma saída tangível de uma atividade prevista em um plano de projeto.
Condição (mostrada como um paralelogramo)	Condição é tanto uma precondição a ser mantida antes que um processo ou uma atividade possa ser iniciado quanto uma pós-condição a ser mantida depois que um processo ou uma atividade tenha terminado.
Papel (mostrado como um círculo com sombreado)	Papel é uma área limitada de responsabilidade. Exemplos de papéis podem ser gerente de configuração, engenheiro de teste e projetista de software. Uma pessoa pode assumir muitos papéis e um único papel pode ser associado a muitas pessoas diferentes.
Exceção (não mostrada nos exemplos aqui, mas pode ser representada como uma caixa de bordas duplas)	Exceção é uma descrição de como modificar o processo caso algum evento previsto ou não previsto ocorra. As exceções são freqüentemente indefinidas, sendo deixadas para desafiar a ingenuidade dos gerentes de projeto e engenheiros.
Comunicação (mostrada como uma seta)	A comunicação é uma troca de informações entre pessoas ou entre pessoas e sistemas computacionais de apoio. As comunicações podem ser formais ou informais. Comunicações formais podem ser a aprovação de uma entrega por um gerente de projeto; comunicações informais poderiam ser a troca de e-mails para resolver ambigüidades em um documento.

Figura 28.5
Módulo de teste do processo.

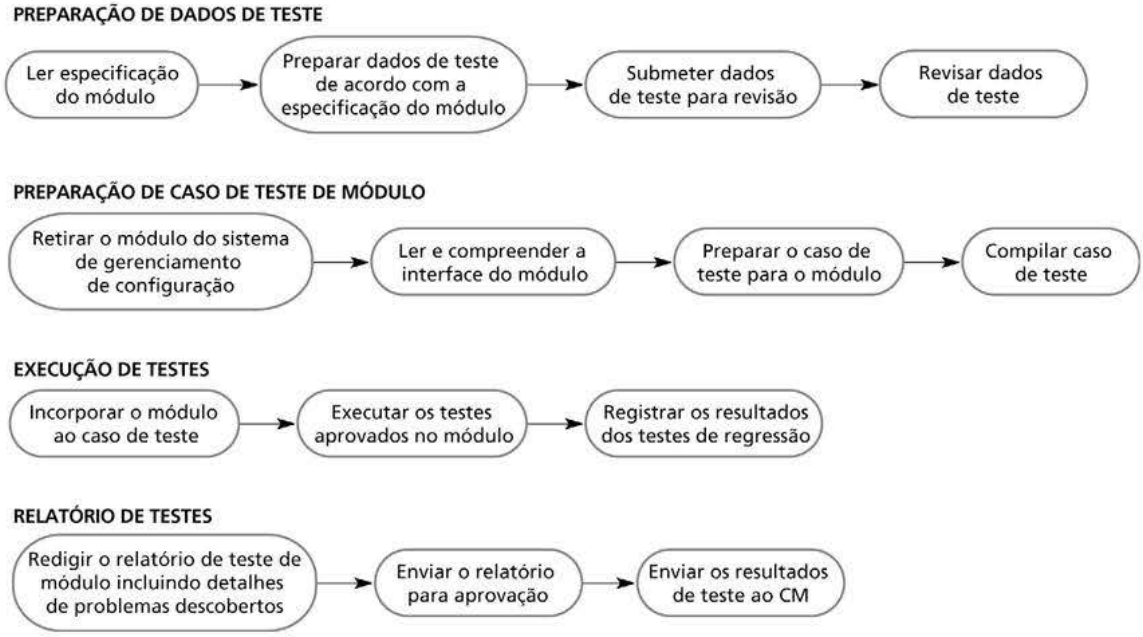


um processo de gerenciamento de configuração controlado rigorosamente (veja o Capítulo 29). O software em teste e os dados do teste estão sob controle de configuração. A Figura 28.5 mostra o papel responsável do processo de teste, as entradas e as saídas do processo e as pré e pós-condições.

A Figura 28.6 mostra o processo 'teste de módulo' como uma série de atividades separadas. Esse fragmento mostra somente as atividades de uma atividade relativamente simples de teste de módulo. Os quatro fluxos de atividades estão relacionados com a preparação de dados de testes, escrita de um caso de teste para o módulo, execução de testes e relato de testes. Essas atividades nos fluxos de preparação são intercaladas. Obviamente, as atividades de preparação precedem as atividades de execução e o relato de testes.

Deixei de fora as informações sobre pré e pós-condições de processo e as entradas e saídas de processo nesse diagrama porque as informações tornariam o modelo mais complexo e difícil de ser compreendido. Em vez de tentar obter todas as informações em um único modelo, pode ser necessário elaborar diversos modelos em diferentes níveis de abstração. Eles devem ser relacionados por meio de elementos comuns, como atividades ou entregas. Alguns modelos devem basicamente estar relacionados com atividades de processo; outros com informações de controle que orientam a execução de processo.

Figura 28.6 Atividades envolvidas no teste de módulo.



28.4.1 Exceções de processos

Processos de software são entidades muito complexas. Quando existe um modelo de processo definido em uma organização, esse modelo pode somente representar uma situação ideal em que a equipe de desenvolvimento ainda não esteve diante de nenhum problema não previsto. Na verdade, problemas imprevistos são fatos da vida diária dos gerentes de projetos. O modelo de processo 'ideal' deve ser modificado dinamicamente quando soluções para esses problemas são encontradas. Exemplos desses tipos de exceções com as quais um gerente de projeto pode lidar incluem:

1. Muitas pessoas-chave que ficam doentes ao mesmo tempo pouco antes de uma revisão crítica de projeto.
2. Abertura séria na proteção de computadores que significa que todas as comunicações estão fora de operação durante muitas horas.
3. Reestruturação organizacional que significa que os gerentes têm de despender muito do seu tempo de trabalho com problemas organizacionais em vez de com gerenciamento de projeto.
4. Solicitação não prevista para redigir propostas para novos projetos que significa que o esforço deve ser transferido do projeto para uma proposta.

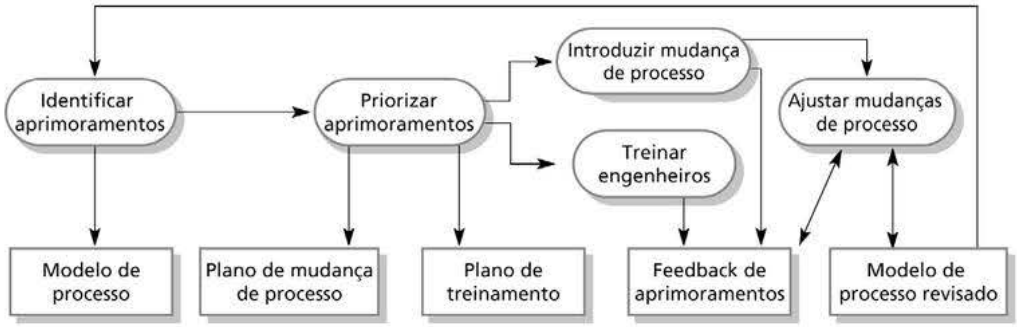
Em geral, o efeito de uma exceção é que, de alguma forma, os recursos, o orçamento ou o cronograma do projeto são alterados. É difícil prever todas as exceções antecipadamente e a sua incorporação no modelo de processo formal. Você tem, portanto, de se esforçar freqüentemente para tratar as exceções e depois, dinamicamente, mudar o processo 'padrão' para lidar com essas circunstâncias inesperadas. Modelos de processos, portanto, são inevitavelmente incompletos e o gerente do processo é responsável por lidar com as exceções e adaptar o processo quando necessário.

28.5 Mudança de processos

Mudança de processo envolve a realização de modificações no processo existente. Você pode fazer isso por meio da introdução de novas práticas, métodos ou ferramentas, mudança de ordem das atividades de processo, introdução ou remoção de entregas de processo ou introdução de novos papéis e responsabilidades. Você deve estabelecer objetivos para o aprimoramento de processos, como 'reduzir o número de defeitos descobertos durante a execução dos testes de integração em 25%'. Esses objetivos devem orientar as mudanças de processo e, depois que as mudanças forem implementadas, são usadas para medir o seu progresso.

Existem cinco estágios-chave no processo de mudança de processo (Figura 28.7):

Figura 28.7 Processo de mudança de processo.



1. *Identificação de aprimoramentos*. Esse estágio está relacionado com o uso dos resultados da análise de processo para identificar a qualidade, cronograma ou gargalos de custo nos quais os fatores de processo poderiam influenciar de maneira adversa a qualidade de produto. O aprimoramento de processos deve enfocar o alívio desses gargalos mediante a proposta de novos procedimentos, métodos e ferramentas para resolver os problemas.
2. *Priorização de aprimoramentos*. Esse estágio está relacionado com a avaliação de possíveis mudanças e com priorizá-las para serem implementadas. Quando muitas mudanças possíveis foram identificadas, é normalmente impossível introduzi-las todas ao mesmo tempo; você precisa decidir quais são as mais importantes. Você pode tomar essas decisões com base na necessidade de aprimorar áreas específicas de processos, os custos para a introdução de mudança, o impacto da mudança na organização e em outros fatores.
3. *Introdução de mudanças de processo*. A introdução de mudanças de processo significa colocar novos procedimentos, métodos e ferramentas nos devidos lugares e integrá-los com outras atividades de processo. Você deve programar o tempo necessário para introduzir mudanças e para assegurar que essas mudanças sejam compatíveis com outras atividades de processo e com os procedimentos e padrões organizacionais.
4. *Treinamento de mudanças de processo*. Sem treinamento não é possível obter benefícios completos das mudanças de processo. Os gerentes de processo e os engenheiros de software podem simplesmente se recusarem a aceitar o novo processo. Muito freqüentemente, as mudanças de processos têm sido impostas sem o adequado treinamento e os efeitos delas acabam degradando a qualidade do produto em vez de aumentá-la.
5. *Ajuste de mudanças*. As mudanças de processo propostas nunca serão completamente eficientes assim que forem introduzidas. Você precisa de uma fase de ajuste em que pequenos problemas são descobertos e as modificações de processo são propostas e introduzidas. Essa fase de ajuste deve durar muitos meses até que os engenheiros de desenvolvimento estejam satisfeitos com o novo processo.

Uma vez que as alterações tenham sido introduzidas, o processo de aprimoramento pode se repetir com análises adicionais para identificar problemas de processo, aprimoramentos propostos etc. É impraticável introduzir muitas mudanças ao mesmo tempo. Independentemente dos problemas de treinamento que isso causa, a introdução de muitas mudanças ao mesmo tempo torna impossível avaliar o efeito de cada mudança no processo.

Como gerente, você precisa estar atento à reação das pessoas da sua equipe quando um processo de mudança for introduzido. A reengenharia de processo de negócio (Hammer, 1990; Ould, 1995), moda da década de 1990 que envolvia realizar mudanças radicais em processos, foi largamente malsucedida porque falhou em levar em conta as considerações legítimas das pessoas envolvidas. As pessoas sentiram que suas habilidades estavam sendo ignoradas e que seu trabalho fora alterado sem consulta prévia. Resistiram ativamente às mudanças e se asseguraram de que as mudanças não funcionariam.

Não há dúvidas de que algumas pessoas se sentem ameaçadas pelas mudanças e consideram que podem perder seu emprego ou ser incapazes de lidar com novas formas de trabalho. Você precisa envolver a equipe de toda maneira no processo de mudança, compreender suas dúvidas e envolvê-la no planejamento do novo processo. Transformando-os em stakeholders da mudança de processo, é muito mais provável que eles desejem que dê certo.

28.6 Framework CMMI de aprimoramento de processos

O Instituto de Engenharia de Software (SEI — *Software Engineering Institute*) foi criado para aumentar as capacitações da indústria de software dos EUA. Em meados da década de 1980, o SEI iniciou um estudo de formas de avaliação de capacitação de fornecedores. O resultado dessa avaliação de capacitação foi o Modelo de Maturidade de

Capacitação (CMM — *Capability Maturity Model*) do SEI (Paulk et al., 1993) (Paulk et al., 1995). Esse modelo exerceu forte influência no convecimento da comunidade de engenharia de software em considerar seriamente o aprimoramento de processos. O CMM para software foi seguido por uma variedade de outros modelos de maturidade de capacitação, entre eles o Modelo de Maturidade de Capacitação de Pessoas (P-CMM — *People Capability Maturity Model*) (Curtis, et al., 2001), explicado no Capítulo 25.

Outras organizações desenvolveram modelos de maturidade de processo semelhantes. A abordagem SPICE para avaliação de capacitação e o aprimoramento de processos (Paulk e Konrad, 1994) é mais flexível do que o modelo do SEI. Ele inclui níveis de maturidade comparáveis aos níveis do SEI, mas também identifica processos, como processos cliente–fornecedor, que atravessam esses níveis. À medida que a maturidade aumenta, o desempenho dos processos principais deve melhorar.

O projeto Bootstrap tinha o objetivo de estender e adaptar o modelo de maturidade do SEI para torná-lo aplicável em uma variedade maior de empresas. O modelo Bootstrap (Haase, et al., 1994; Kuvaja, et al., 1994) usa os níveis de maturidade do SEI e também propõe:

1. Guias para um sistema de qualidade corporativo para apoiar o aprimoramento de processos.
2. Uma importante distinção entre organização, metodologia e tecnologia.
3. Um modelo-base de processo (baseado no modelo usado na Agência Espacial Européia) que pode ser adaptado.

Em uma tentativa de integrar a grande quantidade de modelos que foram desenvolvidos (entre eles os seus próprios modelos), o SEI embarcou em um novo programa para desenvolver um modelo de capacitação integrada (CMMI). Este substitui os CMMs de Engenharia de Sistemas e o de Software e integra outros modelos de engenharia. Tem duas representações, por estágios e contínuo, e resolve alguns pontos fracos do modelo CMM para Software.

O modelo CMMI (Ahern, et al., 2001) tem a intenção de ser um framework de aprimoramento de processos que tem aplicabilidade ampla por meio de uma veriedade de empresas. A versão por estágios é compatível com o CMM para Software e permite aos processos de desenvolvimento e gerenciamento de sistemas de uma organização ser avaliados e que lhe sejam atribuídos um nível de maturidade de 1 a 5. A versão contínua permite uma classificação mais fina e avalia 24 áreas de processo (veja a Tabela 28.3) em uma escala de 1 a 6.

O modelo é muito complexo (mais de mil páginas de descrição), de modo que o simplifiquei radicalmente para explicação nesta seção:

Tabela 28.3 Áreas de processo no CMMI

Categoria	Área de processo
Gerenciamento de processo	Definição de processo organizacional Foco no processo organizacional Treinamento organizacional Desempenho de processo organizacional Inovação e implantação organizacional
Gerenciamento de projeto	Planejamento de projeto Monitoração e controle de projeto Gerenciamento de acordo com fornecedores Gerenciamento de projeto integrado Gerenciamento de riscos Integração de equipes Gerenciamento quantitativo de projeto
Engenharia	Gerenciamento de requisitos Desenvolvimento de requisitos Solução técnica Integração de produto Verificação Validação
Apoio	Gerenciamento de configuração Gerenciamento de qualidade de processo e produto Medição e análise Análise de decisão e resolução Ambiente organizacional para integração Análise causal e resolução

1. *Áreas de processo*. O CMMI identifica 24 áreas de processo relevantes para a capacitação e aprimoramento do processo de software. Elas estão organizadas em quatro grupos no modelo CMMI contínuo. Esses grupos e áreas de processos relacionados estão listados na Tabela 28.3.
2. *Objetivos*. Os objetivos são descrições abstratas de um estado desejado que devem ser atingidos por uma organização. O CMMI possui objetivos específicos que estão associados a cada área de processo e que definem o estado desejável para cada área. O modelo também define objetivos genéricos associados com a institucionalização de boas práticas. A Tabela 28.4 mostra alguns objetivos genéricos e específicos do CMMI.
3. *Práticas*. As práticas no CMMI são descrições de maneiras de se atingir um objetivo. Mais de sete práticas genéricas e específicas podem ser associadas com cada objetivo dentro de cada área de processo. Exemplos de práticas recomendadas são mostrados na Tabela 28.5. No entanto, o CMMI reconhece que o objetivo é mais importante do que a maneira como o objetivo é alcançado. As organizações podem usar quaisquer práticas apropriadas para se atingir qualquer um dos objetivos do CMMI — elas não têm de adotar as recomendações do CMMI.

Objetivos e práticas genéricas não são técnicas, mas estão relacionadas com a institucionalização de boas práticas, o que significa que depende da maturidade da organização. Portanto, em uma organização jovem, nos seus primeiros estágios de desenvolvimento de maturidade, a institucionalização pode significar a adoção de planos e processos estabelecidos. Contudo, para uma organização com processos avançados e mais maduros, a institucionalização pode significar o controle de processos por meio de técnicas estatísticas ou outras técnicas quantitativas.

A avaliação de CMMI envolve o exame dos processos organizacionais e a avaliação dos processos por uma escala de seis pontos que se relaciona ao nível de maturidade em cada área de processo.

Tabela 28.4 Exemplos de objetivos no CMMI

Objetivo	Área de processo
Ações corretivas são gerenciadas até a conclusão quando o desempenho do projeto ou os resultados desviam significativamente do plano.	Objetivo específico na monitoração e controle de projeto
Desempenho real e o progresso do projeto são monitorados contra o plano do projeto.	Objetivo específico na monitoração e controle de projeto
Os requisitos são analisados e validados e uma definição da funcionalidade necessária é desenvolvida.	Objetivo específico no desenvolvimento de requisitos
Causas principais de defeitos e outros problemas são sistematicamente determinados.	Objetivo específico na análise causal e resolução
O processo é institucionalizado como um processo definido.	Objetivo genérico

Tabela 28.5 Práticas e objetivos associados ao CMMI

Prática	Objetivo associado
Analisar os requisitos derivados para assegurar que eles são necessários e suficientes. Validar requisitos para assegurar que os produtos resultantes serão executados conforme esperado no ambiente do usuário usando várias técnicas quando apropriado.	Os requisitos são analisados e validados, e uma definição de funcionalidade necessária é desenvolvida.
Selecionar defeitos e outros problemas para análise. Executar análise causal de defeitos selecionados e outros problemas e ações propostos para resolvê-los.	Causas principais de defeitos e outros problemas são sistematicamente determinados.
Estabelecer e manter uma política organizacional para planejamento e execução do processo de desenvolvimento de requisitos. Atribuir responsabilidade e autoridade para executar o processo, desenvolvendo produtos de trabalho e fornecendo os serviços do processo de desenvolvimento de requisitos.	O processo é institucionalizado como um processo definido.

A escala de seis pontos atribui um nível para um processo como se segue:

1. *Não realizado*. Um ou mais objetivos específicos associados com a área de processo não estão satisfeitos.
2. *Realizado*. Os objetivos associados às áreas de processo estão satisfeitos e para todos os processos o escopo de trabalho a ser realizado está explicitamente estabelecido e comunicado aos membros da equipe.
3. *Gerenciado*. Neste nível, os objetivos associados com as áreas de processo são atendidos e as políticas organizacionais estão estabelecidas de modo que definam quando cada processo deve ser usado. Devem existir planos documentados e os procedimentos de gerenciamento de recursos e de monitoração de processos devem estar estabelecidos por toda a instituição.
4. *Definido*. Este nível enfoca a padronização da organização e a implantação de processos. Cada projeto na organização tem um processo gerenciado configurado com base em um grupo definido de processos organizacionais. Os ativos de processo e as medições de processo coletadas devem ser usados nos aprimoramentos de processos futuros.
5. *Quantitativamente gerenciado*. Neste nível existe uma responsabilidade organizacional para o uso de métodos estatísticos e outros métodos quantitativos para controlar os subprocessos. Isto é, as medições de processo e de produto coletadas devem ser usadas no gerenciamento de processo.
6. *Otimização*. Neste nível mais alto, a organização deve usar as medições de processo e de produto para dirigir o aprimoramento de processo. Tendências devem ser analisadas e os processos devem ser adaptados para as necessidades de mudança de negócios.

Naturalmente, essa descrição dos níveis de capacitação está simplificada, mas você pode pensar nos níveis, de maneira geral, como progressivos, com descrições explícitas de processo atribuídas aos níveis mais baixos por meio da padronização de processo para mudança e aprimoramento de processo dirigidos pelas medições do processo e do software.

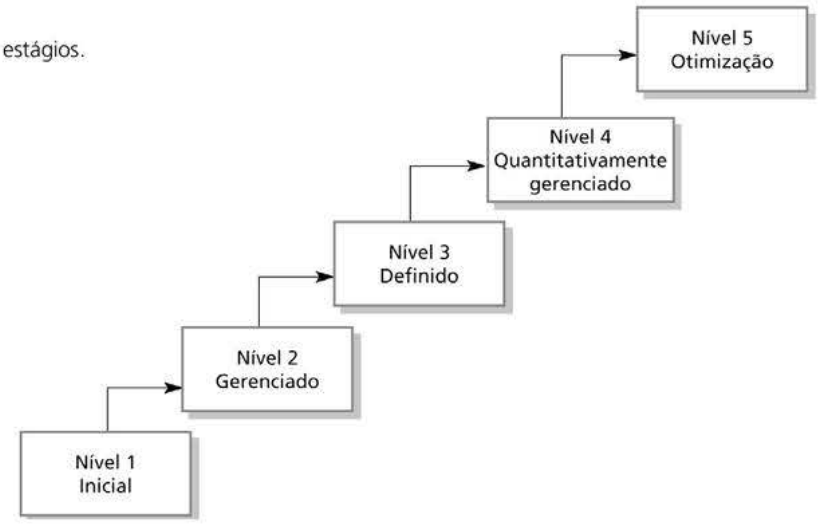
28.6.1 Modelo CMMI por estágios

O modelo CMMI por estágios é comparável com o CMM para Software na medida em que fornece um meio para avaliar a capacitação dos processos de uma organização em um dos cinco níveis. Ele prescreve os objetivos que devem ser alcançados em cada um dos níveis. O aprimoramento do processo é atingido pela implementação de práticas a cada nível, movendo-se dos níveis mais baixos para os mais elevados no modelo. Os cinco níveis do modelo CMMI por estágios são mostrados na Figura 28.8

Cada nível de maturidade tem um conjunto de áreas de processo associadas e objetivos genéricos. Por exemplo, as áreas de processo, conforme definidas no modelo, associadas com o segundo nível (o nível gerenciado) são:

1. *Gerenciamento de requisitos*. Gerenciar os requisitos de produtos e de componentes de produtos do projeto e identificar inconsistências entre esses requisitos e planos e produtos de trabalho do projeto.
2. *Planejamento de projeto*. Estabelecer e manter os planos que definem as atividades de projeto.
3. *Monitoração e controle de projeto*. Provê a compreensão do progresso do projeto de modo que ações corretivas adequadas possam ser tomadas quando o desempenho do projeto se desviar significativamente do plano.

Figura 28.8
Modelo CMMI por estágios.



4. *Gerenciamento de acordo com fornecedores*. Gerenciar a aquisição de produtos e serviços de fornecedores externos para o projeto para os quais existe um acordo formal.
5. *Medição e análise*. Desenvolver e manter uma capacitação em medição usada para apoiar as necessidades de informações gerenciais.
6. *Garantia de qualidade de processo e produto*. Prover pessoal e gerenciamento com visão objetiva dos processos e produtos de trabalho associados.
7. *Gerenciamento de configuração*. Estabelecer e manter a integridade dos produtos usando identificação, controle, relatório de status e auditorias de configuração.

Bem como essas práticas específicas, a organização que opera no segundo nível do modelo CMMI deve ter atingido um objetivo genérico de institucionalização de cada um dos processos conforme um processo gerenciado. Exemplos de práticas institucionais associadas com o planejamento de projeto, que conduzem para o processo de planejamento de projeto, no caso de um processo gerenciado, são:

- Estabelecer e manter uma política organizacional para planejamento e execução do processo de planejamento de projeto.
- Prover recursos adequados para execução do processo de gerenciamento de projeto, para desenvolvimento de produtos e fornecimento de serviços do processo.
- Monitorar e controlar o processo de planejamento de projeto contra o plano e tomar ação corretiva apropriada.
- Revisar as atividades, o status e os resultados do processo de planejamento de projeto com gerenciamento de alto nível, e resolver questões.

A vantagem do CMMI por estágios, independentemente da sua compatibilidade com o CMM para Software, é que ele define um caminho claro de aprimoramento para as organizações. As organizações movem-se do segundo para o terceiro nível e daí por diante. A desvantagem é que pode ser mais adequado introduzir objetivos e práticas de níveis mais elevados antes das práticas de níveis inferiores. Quando uma organização faz isso, a avaliação de maturidade fornecerá um quadro errôneo de sua capacitação.

28.6.2 Modelo CMMI contínuo

Os modelos de maturidade contínua não classificam uma organização em níveis discretos. Pelo contrário, são modelos de granularidade mais baixa que consideram uma prática individual ou grupo de práticas e avaliam o uso de cada prática. A avaliação de maturidade não é, portanto, um valor único, mas um conjunto de valores que mostra a maturidade da organização em cada processo ou em cada grupo de processo.

O CMMI contínuo avalia cada área de processo conforme mostrado na Tabela 28.3 e estabelece um nível de avaliação de capacitação de 1 a 6 (conforme já descrito) para cada área de processo.

Normalmente, as organizações operam em diferentes níveis de maturidade para diferentes áreas de processo. Conseqüentemente, o resultado de uma avaliação em CMMI contínuo é um perfil de capacitação que mostra cada área

Figura 28.9

O processo de mudança do processo

de processo e sua avaliação de capacitação associada. Um fragmento de um perfil de capacitação que mostra processos em diferentes níveis de capacitação é mostrado na Figura 28.9. As organizações podem desenvolver perfis de capacitação reais e objetivos nos quais o perfil objetivo reflete o nível de capacitação para essa área de processo que eles gostariam de atingir.

A principal vantagem do modelo contínuo é que as organizações podem escolher os processos a ser aprimorados de acordo com suas próprias necessidades e requisitos. Na minha experiência, diferentes tipos de organizações possuem diferentes requisitos para o aprimoramento de processos. Por exemplo, uma empresa que desenvolve software para a indústria aeroespacial pode enfocar os aprimoramentos na especificação de sistema, no gerenciamento de configuração e validação, ao passo que uma empresa de desenvolvimento para a Web pode estar mais dedicada a processos de interação com clientes. O modelo por estágios requer que as empresas enfoquem os diferentes estágios por vez. Em contraste, o CMMI contínuo permite mais discrição e flexibilidade enquanto conservam o apoio ao CMMI.

PONTOS-CHAVE

- O aprimoramento de processos envolve a análise de processo, padronização, medição e mudança. O treinamento é essencial para a eficiência do aprimoramento de processos.
- Os processos podem ser classificados como informais, gerenciados, metódicos e de aprimoramento. Essa classificação pode ser usada para identificar o apoio de ferramentas aos processos.
- O ciclo de aprimoramento de processos envolve medição, análise e modelagem, e mudança de processos.
- As medições devem ser usadas para responder a questões específicas sobre o processo de software usado. Essas questões devem basear-se nos objetivos de aprimoramento organizacionais.
- Os três tipos de métricas usadas no processo de medição são métricas de tempo, métricas de uso de recursos e métricas de eventos.
- Modelos de processo incluem descrições de atividades, subprocessos, papéis, exceções, comunicações, entregas e outros processos.
- O modelo de maturidade de processo CMMI é um modelo integrado de aprimoramento de processos que apóia tanto os aprimoramentos de processo por estágios quanto os contínuos.
- O aprimoramento de processos no modelo CMMI baseia-se no alcance de um conjunto de objetivos relacionados às boas práticas da engenharia de software e na descrição, padronização e controle de práticas usadas para atingir esses objetivos. O modelo CMMI inclui práticas recomendadas que podem ser usadas, mas não são obrigatórias.

LEITURAS SUGERIDAS

CMMI distilled. Resumo da abordagem CMMI no momento em que este livro estava sendo escrito. Ele é mais fácil de ler se você já tiver conhecimento do CMM para software porque ele não contém material introdutório geral sobre aprimoramento de processos (D.M. Ahern, et al., 2001, Addison-Wesley.)

"Can you trust software capability evaluations". Este artigo apresenta uma visão cética sobre o assunto de avaliação de capacitação e explica por que essas avaliações podem não fornecer um quadro verdadeiro de maturidade da organização (E. O'Connell e H. Saiedian, *IEEE computer*, 33(2), fevereiro de 2000).

Trends in software: software process modelling and technology. Este livro inclui uma boa seleção de artigos gerais que abrangem aspectos de processos de software, entre eles modelagem de processos, apoio de processo e uso do CMM (A. Fuggetta e A. Wolf (eds.), 1996, John Wiley & Sons.)

Managing the software process. Texto clássico sobre aprimoramento de processos de software que trouxe a noção de uma abordagem estruturada para aprimoramento de processos. Embora este livro tenha mais de 15 anos de existência, suas recomendações são ainda muito relevantes, especialmente para projetos de software de grande porte (W. S. Humphrey, 1988, Addison-Wesley.)

EXERCÍCIOS

28.1 Sugira modelos de processo para os seguintes processos:

- Acender fogo em madeira
- Fazer uma refeição com três pratos (você escolhe o menu)
- Escrever um programa pequeno (50 linhas)

28.2 Sob quais circunstâncias a qualidade do produto provavelmente pode ser determinada pela qualidade da equipe de desenvolvimento? Dê exemplos de tipos de produtos de software particularmente dependentes do talento e da habilidade individuais.

28.3 Explique por que um processo metódico não é necessariamente um processo gerenciado conforme definido na Seção 28.2.

28.4 Sugira três ferramentas especiais de software que possam desenvolvidas para apoiar um programa de aprimoramento de processos em uma organização.

28.5 Suponha que o objetivo de aprimoramento de processos em uma organização seja aumentar o número de componentes reusáveis produzidos durante o desenvolvimento. Sugira três questões no paradigma GQM que possam conduzir a esse objetivo.

28.6 Descreva três tipos de métricas de processos de software que possam ser coletadas como parte de um processo de aprimoramento. Dê um exemplo de cada tipo de métrica.

28.7 Projete um processo para avaliação e priorização de propostas de mudança de processos. Documente esse processo como um modelo mostrando os papéis envolvidos.

28.8 Forneça duas vantagens e duas desvantagens da abordagem para avaliação e aprimoramento de processos incorporada aos frameworks de aprimoramento de processos como o CMMI.

28.9 Sob quais circunstâncias você recomenda o uso da representação por estágios do CMMI?

28.10 Qual é a diferença entre objetivos genéricos e específicos no CMMI?

28.11 Quais são as vantagens e as desvantagens no uso de um modelo de maturidade baseado em objetivos em vez de baseado em práticas?

28.12 Os programas de aprimoramento de processos que envolvem medições do trabalho de pessoas no processo e na mudança do processo são inerentemente desumanos? Qual resistência ao programa de aprimoramento de processo pode surgir?

29

Gerenciamento de configurações

Objetivos

O objetivo deste capítulo é apresentar o processo de gerenciamento de código e documentação no desenvolvimento do sistema de software. Após ler este capítulo, você:

- compreenderá por que o gerenciamento de configurações de software é necessário para sistemas complexos;
- terá sido apresentado a quatro atividades fundamentais de gerenciamento de configurações: planejamento de gerenciamento de configurações, gerenciamento de mudanças, gerenciamento de versões e construção de sistemas;
- compreenderá como as ferramentas CASE (*Computer Aided Software Engineering*) são utilizadas para apoiar os processos de gerenciamento de configurações.

Conteúdo

29.1 Planejamento de gerenciamento de configurações
29.2 Gerenciamento de mudanças
29.3 Gerenciamento de versões e releases
29.4 Construção de sistemas
29.5 Ferramentas CASE para gerenciamento de configurações

O gerenciamento de configuração (CM — *Configuration Management*) é o desenvolvimento e o uso de padrões e procedimentos para o gerenciamento de sistemas de software em desenvolvimento. Como explicado no Capítulo 7, requisitos de sistemas sempre mudam durante o desenvolvimento e o uso, e você deve incorporar esses requisitos às novas versões do sistema. Você deve gerenciar os sistemas em desenvolvimento porque é fácil perder a rastreabilidade de quais mudanças foram incorporadas em qual versão do sistema. As versões incorporam propostas de mudanças, correções de defeitos e adaptações para diferentes hardware e sistemas operacionais. Há muitas versões em desenvolvimento e em uso ao mesmo tempo. Se você não tem procedimentos eficientes de gerenciamento de configuração definidos, pode desperdiçar esforços modificando a versão errada de um sistema, entregar a versão errada de um sistema aos clientes e perder a rastreabilidade de onde o código-fonte do software está armazenado.

Os procedimentos de gerenciamento de configuração definem como registrar e processar mudanças de sistema, como relacioná-las aos componentes do sistema e os métodos usados para identificar

diferentes versões dele. Ferramentas de gerenciamento de configuração são usadas para armazenar versões de componentes do sistema, sistemas construídos com base nesses componentes e rastrear os releases das versões do sistema para os clientes.

O gerenciamento de configuração é considerado algumas vezes parte do gerenciamento da qualidade de software (abordada no Capítulo 27), tendo o mesmo gerente dividindo as responsabilidades do gerenciamento de qualidade e de configuração. O software é inicialmente liberado pela equipe de desenvolvimento para a garantia da qualidade. A equipe de GQ (QA — *Quality Assurance*) verifica se o sistema possui qualidade aceitável. Dessa maneira, ele se transforma num sistema controlado, ou seja, as mudanças que sofrer devem ser acordadas e registradas antes de serem implementadas. Os sistemas controlados são, algumas vezes, chamados de baselines porque são considerados o ponto de partida para futuras evoluções controladas do sistema.

Há muitas razões por que os sistemas existem em diferentes configurações. Configurações podem ser produzidas para diferentes computadores, para diferentes sistemas operacionais, incorporando funções específicas de clientes etc. (Figura 29.1). Os gerentes de configuração são responsáveis por manter a rastreabilidade das diferenças entre versões de software, para assegurar que as novas versões sejam derivadas de maneira controlada e liberar novas versões para clientes certos no momento certo.

A definição e o uso de padrões de gerenciamento de configuração são essenciais para a certificação da qualidade tanto para a o padrão ISO 9000 quanto para os padrões CMM e CMMI (Paulk et al., 1995; Ahern et al., 2002; Peach, 1996). Um exemplo desse tipo de padrão é o IEEE 828-1998, que é um padrão para planos de gerenciamento de configuração. Em uma empresa, esses padrões devem ser incorporados no manual da qualidade ou no guia de gerenciamento de configurações. Naturalmente, padrões genéricos externos podem ser usados como referência para padrões organizacionais detalhados que configuram um ambiente específico.

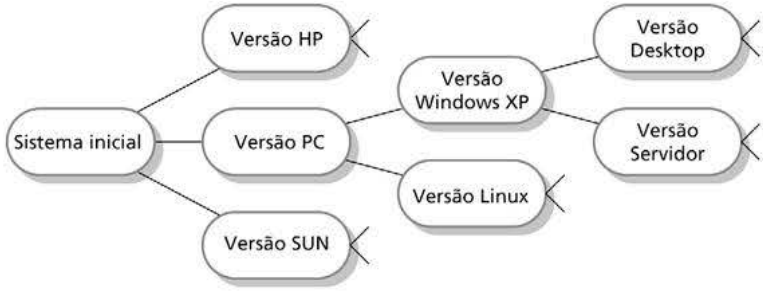
Num processo tradicional de desenvolvimento de software baseado no modelo cascata (veja Capítulo 4), o software é entregue para a equipe de gerenciamento de configuração depois que o desenvolvimento está completo e os componentes individuais de software foram testados. Essa equipe, portanto, assume a responsabilidade pela construção do sistema completo e pelo gerenciamento dos testes do sistema. Defeitos descobertos durante os testes são encaminhados à equipe de desenvolvimento para serem reparados. Após reparados, a equipe de desenvolvimento entrega uma nova versão do componente reparado para a equipe de garantia da qualidade. Se a qualidade for aceitável, ele pode se tornar uma nova baseline para os desenvolvimentos futuros do sistema.

Esse modelo, no qual a equipe de CM controla os processos de integração e teste de sistema, tem influenciado o desenvolvimento de padrões de gerenciamento de configurações. Muitos padrões de CM assumem que o modelo cascata será usado para o desenvolvimento do sistema (Beroff e Davis, 1991). Isso significa que os padrões devem ser adaptados para abordagens modernas de desenvolvimento baseadas em especificações e desenvolvimentos incrementais. Hass (Hass, 2003) explica algumas dessas adaptações para o processo de desenvolvimento de software, como o desenvolvimento ágil.

Para prover um desenvolvimento incremental, algumas organizações desenvolveram uma abordagem modificada para o gerenciamento de configuração que suporta o desenvolvimento atual e o teste de sistema. Essa abordagem recai sobre uma construção muito freqüente (pelo menos diariamente) de todo o sistema com base nos seguintes componentes:

1. A organização de desenvolvimento estabelece determinada hora para a entrega dos componentes do sistema (digamos, 14 horas). Se os desenvolvedores possuírem novas versões dos componentes que eles estão escrevendo, devem entregá-los nesse horário. Os componentes podem estar incompletos mas devem prover algumas funcionalidades básicas que podem ser testadas.
2. Uma nova versão do sistema é construída baseada nesses componentes, pela compilação e ligação deles para formar um sistema completo.
3. Esse sistema é depois entregue para a equipe de testes, que cuida de um conjunto predefinido de testes de sistema. Ao mesmo tempo, os desenvolvedores estão ainda trabalhando nos seus componentes, incluindo funcionalidades e reparando defeitos descobertos nos testes anteriores.

Figura 29.1
Famílias de sistemas.



4. Defeitos descobertos durante os testes de sistema são documentados e devolvidos aos desenvolvedores do sistema. Eles reparam esses defeitos na versão seguinte do componente.

As vantagens do uso de construções diárias do software são que as chances de encontrar problemas oriundos de interações de componentes no processo aumentam. Além disso, construções diárias encorajam testes minuciosos de componentes. Psicologicamente, desenvolvedores são submetidos a pressões para não quebrar a construção, ou seja, entregar uma versão de componente que venha a produzir uma falha completa no sistema. Eles são, portanto, relutantes em entregar versões novas de componentes que não foram adequadamente testados. No mínimo, o tempo de teste de sistema é gasto descobrindo e lidando com defeitos de software que deveriam ter sido encontrados durante os testes de unidade.

O uso bem-sucedido de construções diárias requer um processo de gerenciamento de mudança muito rigoroso para manter a rastreabilidade dos problemas descobertos e reparados. Isso também conduz a um grande número de versões de sistemas e componentes que deve ser gerenciado. Um bom gerenciamento de configuração é, portanto, essencial para que essa abordagem tenha sucesso.

O gerenciamento de configuração nas abordagens de desenvolvimento ágil e rápido não pode estar baseado em procedimentos e burocracias rígidos. Se por um lado estes podem ser necessários para projetos grandes e complexos, por outro lado desaceleram o processo de desenvolvimento. A manutenção cuidadosa de registros é essencial para projetos grandes e complexos desenvolvidos em diferentes localizações, mas é desnecessária para pequenos projetos. Nesses projetos, todos os membros da equipe de trabalho estão juntos numa mesma sala e o custo indireto em manter os registros contribui para desacelerar o processo de desenvolvimento. Contudo, isso não significa que CM deve ser completamente abandonado quando um desenvolvimento rápido for necessário. Preferivelmente, processos ágeis usam ferramentas simples de CM, como as ferramentas de gerenciamento de versão e de construção de sistemas que exigem algum controle. Todos os membros da equipe devem aprender o uso dessas ferramentas e ajustar-se às disciplinas que elas impõem.

29.1 Planejamento de gerenciamento de configurações

O plano de gerenciamento de configurações descreve os padrões e procedimentos que devem ser usados para o gerenciamento. O ponto de partida para o desenvolvimento do plano deve ser um conjunto de padrões de configuração, que devem ser adaptados para se atender aos requisitos e às restrições de cada projeto específico. O plano CM deve ser organizado em um número de seções que:

1. Define o que será gerenciado (itens de configuração) e o esquema que se deve usar para identificar essas entidades.
2. Estabelece quem é responsável pelos procedimentos de gerenciamento de configuração e pela submissão de itens controlados para a equipe de gerenciamento de configuração.
3. Define as políticas de gerenciamento de configuração que todos os membros da equipe devem adotar para o controle de mudanças e gerenciamento de versões.
4. Especifica as ferramentas que se deve usar para o gerenciamento de configurações e os processos para uso dessas ferramentas.
5. Descreve a estrutura do banco de dados de configuração usada para registrar as informações de configuração e as informações que devem ser mantidas nesse banco de dados (os registros de configuração).

Você pode incluir outras informações no plano CM, como o gerenciamento de software oriundo de fornecedores externos e procedimentos de auditoria dos processos de CM.

Uma parte importante do plano CM é a definição de responsabilidades. O plano deve definir quem é responsável pela entrega de cada documento ou de componente de software para a garantia da qualidade e para o gerenciamento de configuração. Também pode definir os revisores de cada documento. A pessoa responsável pela entrega do documento não precisa ser a mesma pessoa responsável por produzir o documento. Para identificar as interfaces, os gerentes de projetos ou os líderes de equipe são freqüentemente responsáveis por todos os documentos produzidos pela sua equipe.

29.1.1 Identificação de item de configuração

Em um grande sistema de software, pode haver módulos de milhares de códigos-fonte, scripts de testes, documentos de projeto etc. Eles são produzidos por pessoas diferentes e, quando criados, podem ser denominados com nomes similares ou idênticos. Para manter a rastreabilidade de todas essas informações de maneira que o arquivo certo possa ser encontrado quando for necessário, você necessita de um esquema de identificação consistente para todos os itens no sistema de gerenciamento de configuração.

Durante o processo de planejamento de gerenciamento de configuração, você decide exatamente quais itens (ou classes de itens) serão controlados. Documentos ou grupos de documentos relacionados em controle de configuração são documentos formais ou itens de configuração. Planos de projeto, especificações, projetos, programas e massa de dados de teste são normalmente mantidos como itens de configuração. Todos os documentos que podem ser úteis para a evolução futura do sistema devem ser controlados pelo sistema de gerenciamento de configuração.

Entretanto, isso não significa que todos os documentos ou arquivos produzidos devem ser colocados sob o controle de configuração. Documentos técnicos de trabalho, que apresentam uma idéia momentânea para um desenvolvimento novo, anotações de reuniões de grupo, esboços de projetos e de propostas, entre outros documentos, não podem ter relevância no longo prazo e não são necessários para a manutenção futura do sistema.

O esquema de identificação de itens de configuração deve atribuir um único nome para todos os documentos sob o controle de configuração. Esse nome pode refletir o tipo do item, uma parte do sistema ao qual ele se aplica, o criador do item etc. No esquema de atribuição de nomes, você pode desejar evidenciar a relação entre os itens para garantir que os documentos relacionados possuam uma mesma raiz em seus nomes. Portanto, você pode definir um esquema de hierarquia com nomes como:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE
PCL-TOOLS/EDIT/HELP/QUERY/HELPFRAMES/FR-1

A parte inicial do nome é o nome do projeto, PCL-TOOLS. Nesse projeto, há algumas ferramentas distintas em desenvolvimento, assim, o nome da ferramenta (EDIT) é usado como a próxima parte do nome. Cada ferramenta inclui módulos denominados diferentemente e cujos nomes definem o próximo componente do identificador do item (FORMS, HELP). O processo de decomposição continua até quando o nível-base de documentos formais é referenciado (Figura 29.2). As folhas da hierarquia de documentos são os itens formais de configuração. A Figura 29.2 mostra que três itens formais são necessários para cada componente de código: uma descrição de objeto (OBJECTS), o código-fonte do componente (CODE) e um conjunto de testes para esse componente (TESTS). Itens como telas de auxílio são também gerenciados e têm diferentes nomes (FR-1, no exemplo anterior).

Esquemas de nomes hierarquizados são simples e de fácil compreensão; algumas vezes, podem mapear uma estrutura de diretórios usada para armazenar arquivos de projeto. No entanto, refletem a estrutura de projeto na qual o software foi desenvolvido. Os nomes de itens de configuração associam componentes com um projeto específico e, dessa maneira, reduzem as oportunidades para o reuso. Pode ser muito difícil encontrar componentes relacionados (por exemplo, todos os componentes desenvolvidos pelo mesmo programador) nos quais a relação não é refletida no esquema item–nome.

29.1.2 Banco de dados de configuração

O banco de dados de configuração é utilizado para registrar todas as informações relevantes sobre as configurações de sistema e os itens de configuração. Você usa o banco de dados CM para auxiliar a avaliação do impacto das mudanças de sistema e para gerar relatórios para a gerência sobre o processo de CM. Como parte do processo de CM, deve-se definir o esquema do banco de dados de CM, os formulários para coletar informações para serem registradas no banco de dados e procedimentos para registro e recuperação de informações de projeto.

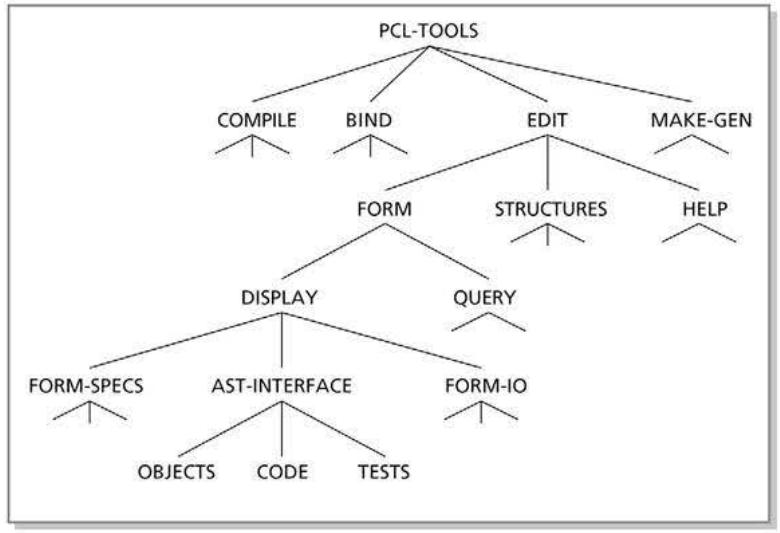


Figura 29.2
Hierarquia da configuração usada para designar identificadores de itens.

Um banco de dados de configuração não inclui apenas informações sobre itens de configuração. Pode também registrar informações sobre usuários de componentes, clientes de sistemas, plataformas de execução, mudanças propostas etc. Pode ser capaz de fornecer respostas para uma variedade de questões sobre configurações de sistemas. Questões típicas podem ser:

1. Quais clientes pegaram um produto de uma versão específica de sistema?
2. Qual configuração de hardware e de sistema operacional é necessária para rodar determinada versão do sistema?
3. Quantas versões de um sistema foram criadas e quais eram as datas da sua criação?
4. Quais versões de um sistema podem ser afetadas se um componente específico for alterado?
5. Quantas solicitações de mudança estão pendentes para uma versão específica?
6. Quantos defeitos reportados existem em uma versão específica?

De preferência, um banco de dados de configuração deve ser integrado com a versão do sistema de gerenciamento usada para armazenar e gerenciar os documentos formais do projeto. Essa abordagem, apoiada por algumas ferramentas CASE integradas, torna possível ligar as mudanças diretamente com os documentos e componentes afetados por elas. Links entre documentos (como documentos de projeto) e códigos de programa podem ser mantidos, de modo que você possa encontrar tudo que tem para modificar quando uma mudança for proposta.

No entanto, ferramentas CASE integradas para o gerenciamento de configurações são caras. Muitas empresas não as usam, mas mantêm o banco de dados de configuração separado dos sistemas de controle de versão. Elas armazenam itens de configuração, como arquivos, numa estrutura de diretórios ou num sistema de gerenciamento de versão como CVS (Berliner, 1990), explicado adiante neste capítulo.

Um banco de dados de configuração armazena informações sobre itens de configuração e referencia seus nomes num sistema de gerenciamento de versão ou depósito de arquivos. Enquanto essa é uma abordagem relativamente barata e flexível, o problema nesse caso é que os itens de configuração podem ser alterados sem passar pelo banco de dados de configuração. Portanto, você não pode estar completamente seguro de que o banco de dados de configuração esteja atualizado com uma descrição atualizada do estado do sistema.

29.2 Gerenciamento de mudanças

Mudanças são um fato da vida para sistemas grandes de software. Conforme expliquei nos capítulos anteriores, as necessidades e requisitos organizacionais alteram-se durante a vida útil de um sistema. Isso significa que você precisa fazer as mudanças correspondentes no sistema de software. Para garantir que essas mudanças serão aplicadas ao sistema de maneira controlada, você precisa de um conjunto de procedimentos de gerenciamento de mudança apoiado por ferramentas.

Procedimentos de gerenciamento de mudança dizem respeito à análise de custo e benefício das mudanças propostas, à aprovação das mudanças viáveis e à rastreabilidade de quais componentes do sistema foram alterados. O processo de gerenciamento de mudança (Figura 29.3) deve surtir efeito quando o software ou a documentação associada são colocados em baseline pela equipe de gerenciamento de configurações.

O primeiro estágio no processo de gerenciamento de configurações é completar um formulário de solicitação de mudança (CRF — *Change Request Form*) que descreva a mudança necessária para o sistema. Da mesma maneira que o registro da mudança é necessário, o CRF registra as recomendações sobre a mudança, os custos estimados e as datas de quando ela foi solicitada, aprovada, implementada e validada. O CRF pode também incluir uma seção em que um analista descreva como a mudança será implementada.

Um exemplo de um formulário de solicitação de mudança parcialmente preenchido é mostrado na Figura 29.4. O formulário de solicitação é geralmente definido durante o processo de planejamento de CM. Esse é um exemplo de um CRF que pode ser usado em projetos de engenharia de sistemas grandes e complexos. Em projetos pequenos, recomendo que as solicitações de mudança sejam formalmente registradas, mas o CRF deve enfocar mais a descrição da mudança necessária e menos os aspectos de implementação. O engenheiro que faz a mudança decide como implementá-la nessas situações.

Uma vez que um formulário de solicitação de mudança é enviado, ele deve ser registrado no banco de dados de configuração. A solicitação de mudança é então analisada para verificar se a mudança solicitada é necessária. Algumas solicitações de mudança podem ser devidas a mal-entendidos, e não a defeitos de sistema; nesse caso, nenhuma mudança no sistema é necessária. Outras podem se referir a defeitos conhecidos. Se a análise descobre que a solicitação é inválida, duplicada ou já foi considerada, a mudança é rejeitada. Você deve comunicar à pessoa que enviou a solicitação de mudança por que ela foi rejeitada.

Figura 29.3

Processo da gestão de alteração.

```
Solicita a alteração mediante o preenchimento do formulário de solicitação de alteração
Analisa a solicitação de alteração
If alteração é válida then
    Acessa como a alteração poderá ser implementada
    Acessa a alteração de custo
    Registra a solicitação no banco de dados
    Submete a solicitação ao controle de alteração
    If a alteração é aceita then
        repeat
            realize a alteração no software
            registre as alterações e link com a solicitação de alteração associada
            submete a alteração do software para a aprovação da qualidade
        até que a qualidade do software esteja adequada
        crie uma nova versão do sistema
    else
        rejeite a solicitação de alteração
else
    rejeite a solicitação da alteração
```

Figura 29.4

Formulário de solicitação de mudança parcialmente preenchido.

Formulário de Solicitação de Mudança

Projeto: Proteus/Ferramenta PCL **Número:** 23/02
Solicitante da mudança: I. Sommerville **Data:** 1/12/02
Mudança solicitada: Quando um componente é selecionado da estrutura, apresentar o nome do arquivo onde ele está armazenado.

Analista da mudança: G. Dean **Data da análise:** 10/12/02
Componentes afetados: Display-Icon.Select, Display-Icon.Display

Componentes associados: FileTable

Avaliação da mudança: Relativamente simples de implementar se uma tabela de nome de arquivo estiver disponível. Requer o projeto e a implementação de um campo na tela. Não é requerida nenhuma mudança dos componentes associados.
Prioridade da mudança: Baixa
Implementação da mudança:
Esforço estimado: 0,5 dia
Data para o CCB: 15/12/02 **Data de decisão do CCB:** 1/2/03
Decisão do CCB: Aceita a mudança. Mudança a ser implementada no Release 2.1
Implementador da mudança: **Data da mudança:**
Data de submissão ao GQ: **Decisão do GQ:**
Data da submissão ao CM:
Comentários

Para mudanças válidas, o estágio seguinte no processo é a avaliação da mudança e o custo. O impacto da mudança no restante do sistema deve ser verificado. Isso envolve a identificação de todos os componentes afetados pela mudança usando informações do banco de dados de configuração e do código-fonte do software. Se realizar a mudança significa que mudanças adicionais em alguma parte do sistema são necessárias, isso aumenta claramente o custo de sua implementação. Em seguida, as mudanças necessárias para os módulos do sistema são avaliadas. Finalmente, o custo para realizar a mudança é estimado, considerando os custos de mudança nos componentes relacionados.

O comitê de controle de mudanças (CCB — *Change Control Board*) deve revisar e aprovar todas as solicitações de mudanças, a menos que as alterações simplesmente envolvam correções de erros menores em telas de displays, páginas Web ou em documentos. O CCB considera o impacto da mudança mais do ponto de vista técnico do que do estratégico e organizacional. O comitê deve decidir se a mudança é economicamente justificada e priorizar as mudanças aceitas.

O termo *comitê de controle de mudanças* nomeia um grupo importante que toma as decisões de mudanças. Os CCBs formalmente estruturados, incluindo profissionais seniores do cliente e do fornecedor, são obrigatórios em projetos militares.

Figura 29.5
Informação de cabeçalho de componente.

```
// Projeto BANKSEC (IST 6087)
//
// BANKSEC-TOOLS/AUTH/RBAC/USER_ROLE
//
// Objeto: currentRole
// Autor: N. Perwaiz
// Data de criação: 10 de novembro de 2002
//
// (c) Lancaster University 2002
//
// Histórico de modificação
// Versão  Implementador  Data       Mudança          Razão
//1.0      J. Jones      1/12/2002  Adicionar cabeçalho Submetido ao CM
//1.1      N. Perwaiz    9/4/2003   Novo campo       Solicit. de mud. R07/02
```

No entanto, para projetos pequenos e médios, o CCB pode simplesmente ser constituído pelo gerente de projeto e por mais um ou dois engenheiros que não estejam diretamente envolvidos no desenvolvimento do software. Em alguns casos, o CCB pode ser um único revisor que manifesta se as mudanças são justificáveis.

O gerenciamento de mudança de pacotes de software genéricos, em vez de sistemas desenvolvidos sob encomenda para determinado cliente, precisa ser tratado de maneira ligeiramente diferente. Nesses sistemas, o cliente não é diretamente envolvido, de modo que a relevância da mudança para o negócio do cliente não é um problema. As solicitações de mudança nesses produtos são geralmente associadas com defeitos no sistema descobertos durante os testes de sistema ou pelos clientes depois que o software foi entregue. Os clientes podem usar uma página Web ou e-mail para relatar os defeitos. A equipe de gerenciamento de defeitos então verifica se os relatórios são válidos e os traduzem para solicitações formais de mudança de sistema. Como em outros tipos de sistemas, as mudanças têm que ser priorizadas para sua implementação e os defeitos podem não ser reparados se os custos de reparo forem muito altos.

Durante o desenvolvimento, quando novas versões do sistema são criadas por meio de construções diárias (ou mais freqüentes) de sistemas, é usado um processo de gerenciamento de mudanças mais simples. Problemas e mudanças devem ainda ser registrados, mas as mudanças que afetam somente componentes e módulos individuais não necessitam ser avaliadas independentemente. Elas são passadas diretamente para o desenvolvedor de sistema. O desenvolvedor do sistema pode aceitá-las ou justificar por que elas não são necessárias. As mudanças que afetam os módulos de sistema produzidos por equipes diferentes de desenvolvimento, no entanto, devem ser avaliadas por uma autoridade que controla as mudanças e as priorize para a implementação.

Em alguns métodos mais ágeis, como extreme programming, os clientes são diretamente envolvidos na decisão se uma mudança for implementada. Quando eles propõem uma mudança para os requisitos do sistema, trabalham com a equipe para avaliar o impacto dela e, então, decidem se a mudança deve ter prioridade, em detrimento das outras características planejadas para o próximo incremento do sistema. No entanto, as mudanças que estão relacionadas ao aprimoramento do software são decididas pelos programadores que trabalham no sistema. A reconstrução, na qual o software é continuamente aprimorado, não é vista como um overhead, mas como uma parte necessária do processo de desenvolvimento.

Na medida em que componentes de software são alterados, um registro de mudanças realizadas em cada componente deve ser mantido. Isso é algumas vezes chamado de *procedência histórica* de um componente. Uma boa maneira de manter a procedência histórica de um componente é fazer um comentário padronizado no início do código-fonte do componente (veja a Figura 29.5). Esse comentário deve referenciar a solicitação de mudanças que disparou a mudança do software. Você pode então escrever scritps simples que varrem todos os componentes e processam as procedências históricas para produzir relatórios de mudança de componentes. Uma abordagem similar pode ser usada para as páginas Web. Para documentos publicados, os registros de mudanças incorporadas em cada versão são usualmente mantidos em uma página separada na frente do documento.

29.3 Gerenciamento de versões e releases

Os processos envolvidos no gerenciamento de versões e releases preocupam-se com a identificação e a manutenção da rastreabilidade das versões de um sistema. Gerentes de versões idealizam procedimentos para assegurar que as versões

de um sistema possam ser recuperadas quando solicitadas e não sejam alteradas acidentalmente pela equipe de desenvolvimento. Para produtos, os gerentes de versões trabalham com a equipe de marketing, e, para sistemas feitos sob encomenda, com os clientes, para planejar quando novos releases de um sistema devem ser criados e distribuídos para implantação.

Uma versão de sistema é uma instância de um sistema que difere, de alguma maneira, de outras instâncias. Versões de sistema podem ter funcionalidades distintas, desempenhos aprimorados ou defeitos de software reparados. Algumas versões podem ser funcionalmente equivalentes, mas projetadas para diferentes configurações de hardware e software. Versões com somente pequenas diferenças são algumas vezes chamadas de *variantes*.

Um release do sistema é uma versão distribuída aos clientes. Cada release deve incorporar novas funcionalidades ou ser planejado para uma plataforma diferente de hardware. Há normalmente muito mais versões de um sistema do que liberações. As versões são criadas no âmbito da organização, para desenvolvimentos ou testes internos, e não são previstas para serem liberadas para os clientes.

Conforme explicado na Seção 29.5, ferramentas CASE são atualmente sempre usadas para apoiar o gerenciamento de versões. Essas ferramentas gerenciam o armazenamento de cada versão de software e controlam o acesso aos componentes de sistema. Os componentes devem ser verificados fora do sistema para a sua edição. O check-in do componente cria uma nova versão e um identificador é definido pelo sistema de gerenciamento de versões. Enquanto as ferramentas são obviamente significativamente diferentes quanto às características que oferecem e quanto às interfaces do usuário, os princípios gerais do gerenciamento de versões abrangidos aqui são a base para todas as ferramentas de apoio.

29.3.1 Identificação de versões

Para criar uma versão específica de um sistema, você precisa especificar as versões dos componentes de sistema que devem ser incluídas nele. Em sistemas grandes de software, há centenas de componentes de software; cada um pode existir em várias versões diferentes. Portanto, deve haver uma maneira não ambígua de identificar cada versão de componente a fim de assegurar que os componentes certos estão incluídos no sistema. Entretanto, você não pode usar o nome do item de configuração para a identificação da versão porque pode haver muitas versões de cada item de configuração identificado.

De outro modo, há três técnicas básicas usadas para a identificação da versão de componente:

1. *Numeração de versões.* O componente recebe um número explícito e único de versão. Isso é o mais comumente usado no esquema de identificação.
2. *Identificação baseada em atributos.* Cada componente tem um nome (como o nome do item de configuração, que não é único para diferentes versões) e um grupo de atributos associados para cada versão (Estublier e Casallas, 1994). Componentes são, portanto, identificados pela especificação de seus nomes e valores de atributos.
3. *Identificação orientada a mudanças.* Cada componente é denominado como na identificação baseada em atributos, mas é também associado com uma ou mais solicitações de mudanças (Munch et al., 1993). Ou seja, considera-se que cada versão de componente foi criada em resposta a uma ou mais solicitações de mudanças. A versão de componente é identificada pelo conjunto de solicitações de mudanças que se aplicam ao componente.

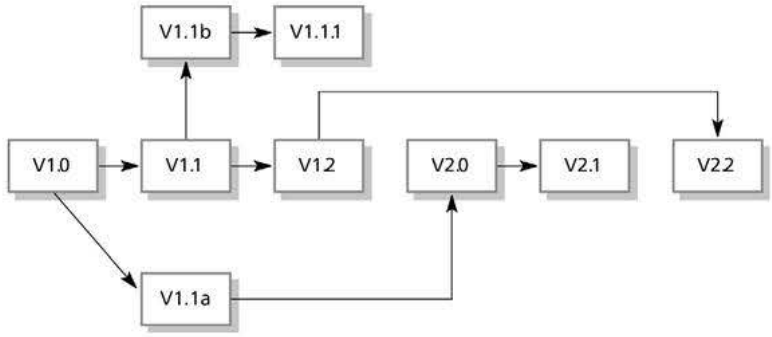
Numeração de versões

Num esquema simples de numeração de versões, um número de versão é adicionado ao componente ou ao nome do sistema. Portanto, você pode se referir ao Solaris 4.3 (versão 4.3 do sistema Solaris) e versão 1.4 do componente **getToken**. Se a primeira versão for chamada de 1.0, as versões subseqüentes serão 1.1, 1.2 e assim por diante. Em algum estágio, um novo release é criado (versão 2.0), e o processo inicia-se novamente a partir da versão 2.1. O esquema é linear, baseado no conceito de que as versões do sistema são criadas em seqüência. Muitas ferramentas de gerenciamento de versões (veja Seção 29.5), como RCS (Tichy, 1985) e CVS (Berliner, 1990), apóiam essa abordagem de identificação.

Essa abordagem e a derivação de um número de versões diferentes de sistema estão ilustradas na Figura 29.6. As setas no diagrama apontam da versão original para a nova versão criada a partir daquela de origem. Observe que a derivação das versões não é necessariamente linear e as versões com números seqüenciais podem ser produzidas com baselines diferentes. Por exemplo, na Figura 29.6, a versão 2.2 é criada da versão 1.2, em vez de da versão 2.1. A princípio, qualquer versão existente pode ser utilizada como ponto de partida para uma nova versão do sistema.

Esse esquema é simples, mas você necessita ter uma quantidade grande de informações extras para manter a rastreabilidade das diferenças entre versões e dos relacionamentos entre propostas de mudança de sistema e as versões. Por exemplo, as versões 1.1 e 1.2 de um sistema podem diferir porque a versão 1.2 foi produzida com a utilização de uma biblioteca de gráficos diferentes. O nome não conta sobre a versão ou por que ela foi criada. Conseqüentemente, você necessita manter registros no banco de dados de configuração que descrevam cada versão e por que ela foi produzida. Você pode também precisar ligar explicitamente as solicitações de mudança para as diferentes versões de cada componente.

Figura 29.6
Estrutura de derivação de versões.



Identificação baseada em atributos

Um problema fundamental com os esquemas explícitos de nomeação de versões é que eles não refletem os diversos atributos que podem ser usados para identificar as versões. Exemplos desses atributos de identificação são:

- Cliente
- Linguagem de desenvolvimento
- Status de desenvolvimento
- Plataforma de hardware
- Data de criação

Se cada versão é identificada por um único conjunto de atributos, é fácil adicionar novas versões derivadas de algumas das versões existentes. Elas são identificadas usando um conjunto único de valores de atributos. Compartilham muitos desses valores com sua versão paterna, de modo que a relação entre as versões é mantida. Você pode recuperar versões específicas pela especificação dos valores de atributos necessários. Funções sobre atributos apóiam consultas, como 'a versão mais recentemente criada' ou 'a versão criada entre duas datas'.

Por exemplo, a versão do sistema de software AC3D desenvolvida em Java para Windows XP em janeiro de 2003 pode ser identificada como:

AC3D (linguagem = Java, plataforma = XP, data = Jan2003)

Usando uma especificação geral de componentes no AC3D, a ferramenta de gerenciamento de versões seleciona as versões de componentes que têm atributos 'Java', 'XP' e 'Jan2003'.

A identificação baseada em atributos pode ser implementada diretamente pelo sistema de gerenciamento de versões, com atributos dos componentes mantidos em um banco de dados do sistema. Alternativamente, o sistema de identificação de atributos pode ser construído como uma camada no topo de um esquema oculto de numeração de versão. O banco de dados de configuração mantém a ligação entre os atributos de identificação e as versões de sistema e componentes considerados.

Identificação orientada a mudanças

A identificação baseada em atributos das versões do sistema remove alguns dos problemas de recuperação de versões de esquemas simples de numeração de versão. No entanto, para recuperar uma versão, você ainda precisa conhecer seus atributos associados. Além disso, você deve usar um sistema de gerenciamento de mudanças separado para descobrir os relacionamentos entre versões e alterações.

A identificação orientada a mudanças é usada para identificar versões de sistemas em vez de componentes. Os identificadores de versões de componentes individuais ficam ocultos dos usuários do sistema CM. Cada mudança de sistema que tenha sido implementada tem um conjunto de mudanças associadas que descreve as mudanças solicitadas realizadas em diferentes componentes do sistema. Os conjuntos de mudanças podem ser aplicados em seqüência, de modo que em princípio, pelo menos, a versão possa incorporar um conjunto arbitrário de mudanças. Por exemplo, um conjunto de mudanças em um sistema feitas para adaptá-lo para Linux em vez de Solaris poderia ser aplicado, seguido das mudanças necessárias para incorporar um novo banco de dados do sistema. Igualmente, as mudanças no Linux/Solaris poderiam ser seguidas por alterações que converteriam para a linguagem de interface de usuário do inglês para o italiano.

Na prática, naturalmente, não é possível aplicar conjuntos arbitrários de mudanças em um sistema. Os conjuntos de mudanças podem ser incompatíveis, de modo que a aplicação de um conjunto de mudanças A seguido pelo conjunto de mudanças D pode criar um sistema inválido. Além disso, os conjuntos de mudanças podem conflitar quando mudanças diferentes afetam o mesmo código do sistema. Se o código foi alterado pelo conjunto de mudanças A, o conjunto

de mudanças D pode deixar de ser útil. Para resolver essas dificuldades, ferramentas de gerenciamento de versão que apóiam a identificação orientada a mudanças permitem que regras de consistência de sistema sejam especificadas. Elas limitam a maneira pela qual os conjuntos de mudanças podem ser combinados.

29.3.2 Gerenciamento de releases

Um release de sistema é uma versão do sistema distribuído para os clientes. Os gerentes de releases de sistemas são responsáveis por decidir quando um sistema pode ser liberado para os clientes, gerenciar o processo de criação do release e de meios de distribuição e documentar o release para assegurar que ele pode ser recriado exatamente como foi distribuído, se for necessário.

Um release de sistema não é somente um código executável do sistema. Ele pode incluir:

1. *Arquivos de configuração* que definem como o release pode ser configurado para instalações específicas.
2. *Arquivo de dados* necessários para a operação do sistema com sucesso.
3. *Um programa de instalação* usado para auxiliar a instalação do sistema no hardware-alvo.
4. *Documentação eletrônica e em papel* que descreve o sistema.
5. *Empacotamento e publicidade associada* projetados para release.

Os gerentes de release não podem considerar que os clientes sempre instalarão releases novos de sistema. Alguns usuários de sistemas podem se sentir satisfeitos com um sistema existente. Eles podem considerar que não vale a pena o custo de alteração para um novo release. Novos releases de sistema não podem, portanto, confiar na instalação de releases anteriores. Para ilustrar esse problema, considere o seguinte cenário:

1. O release 1 de um sistema é distribuído e colocado em uso.
2. O release 2 requer a instalação de arquivos novos de dados, mas alguns clientes não necessitam de recursos do release 2 e, desta maneira, permanecem com o release 1.
3. O release 3 requer arquivos de dados instalados no release 2 e não tem arquivos novos de dados.

O distribuidor de software não pode presumir que os arquivos requeridos pelo release 3 tenham sido instalados em todos os sites. Alguns sites podem ir diretamente do release 1 para o 3, pulando o release 2. Outros podem ter modificado os arquivos de dados associados com o release 2 para refletir circunstâncias locais. Portanto, os arquivos de dados devem ser distribuídos e instalados com o release 3 do sistema.

Tomada de decisão para um release

A preparação e a distribuição de um release de sistema é um processo dispendioso, particularmente para produtos de software voltados para um grande mercado. Se os releases forem muito freqüentes, os clientes poderão deixar de atualizar os antigos releases, especialmente se os novos não forem gratuitos. Se os releases de sistemas não forem freqüentes, poderá haver perda de participação no mercado, à medida que os clientes migram para sistemas alternativos. Isso, naturalmente, não se aplica ao software desenvolvido particularmente para determinada organização. Para esse tipo de software, releases não freqüentes podem significar aumento da divergência entre o software e o processo de negócio que foi projetado para apoiá-lo.

Vários fatores técnicos e organizacionais que você deve levar em consideração quando da decisão de criar um novo release de sistema são mostrados na Tabela 29.1.

Criação de releases

A criação de um release é um processo de criação de arquivos e documentos que inclui todos os componentes do release do sistema. O código executável de programas e todos os arquivos de dados associados devem ser coletados e identificados. As descrições de configuração podem ter que ser escritas para hardwares diferentes e para instruções e sistemas operacionais preparados para clientes que necessitem configurar os próprios sistemas. Se os manuais a serem lidos em computadores são distribuídos, cópias eletrônicas devem ser armazenadas com o software. Scripts para a instalação do programa podem ser escritos. Finalmente, quando todas as informações estiverem disponíveis, o diretório do release é manipulado para a distribuição.

O meio normal de distribuição para releases de sistema é agora o disco óptico (CD-ROM ou DVD) que pode armazenar de 600 Mbytes a 4 Gbytes de dados. Adicionalmente, o software pode ser liberado on-line, permitindo que os clientes baixem o arquivo da Internet, embora muitas pessoas considerem esse procedimento longo para o caso de grandes arquivos e prefiram a distribuição em CD-ROM.

Há custos muito altos de marketing e de empacotamento associados com a distribuição de novos releases de produtos de software; assim, os vendedores de produtos de software usualmente criam novos releases somente para novas plataformas ou para adicionar novas funcionalidades significativas. Eles, então, cobram dos usuários o novo software. Quando são des-

Tabela 29.1 Fatores que influenciam a estratégia de liberação de sistema

Fator	Descrição
Qualidade técnica do sistema	Se defeitos sérios de sistema, que afetam a forma como muitos clientes o usam, são comunicados, pode ser necessário criar um release de reparo de defeito. Entretanto, defeitos pequenos de sistema podem ser reparados pela criação de patches (freqüentemente distribuídos pela Internet) que podem ser aplicados no release atual do sistema
Mudanças de plataforma	Você pode criar um novo release de uma aplicação de software quando uma nova versão da plataforma do sistema operacional é liberada.
A quinta lei de Lehman (veja o Capítulo 21)	Ela sugere que o incremento da funcionalidade que é incluída em cada release é aproximadamente constante. Portanto, um release de sistema com funcionalidades novas significantes poderia ser seguido por um release de reparo.
Competição	Um release novo de sistema pode ser necessário porque uma competição de produtos está em curso.
Requisitos de marketing	O departamento de marketing de uma organização pode ter se comprometido a disponibilizar os releases numa determinada data.
Propostas de mudanças do cliente	Para sistemas sob encomenda, os clientes podem ter feito e pago por um conjunto específico de propostas de mudanças de sistema, e eles esperam um release de sistema tão logo quanto elas forem implementadas.

cobertos problemas em um release existente, os vendedores usualmente criam patches para reparar o software disponível em um site Web para ser baixado pelos clientes.

Independentemente dos custos para encontrar e baixar o novo release, o problema é que muitos clientes podem nunca descobrir a existência desses reparos ou podem não ter conhecimento técnico para instalá-los. Eles podem, ao contrário, continuar usando os seus sistemas atuais e com defeitos, com os riscos conseqüentes para os seus negócios. Em algumas situações, nas quais o patch é projetado para reparar pontos fracos de proteção, o risco de falhas na instalação do patch pode significar que o negócio é suscetível a ataques externos.

Documentação de releases

Quando um release de sistema é produzido, ele deve ser documentado para assegurar que possa ser recriado *ipsis literis* no futuro. Isso é particularmente importante para sistemas embutidos de ciclo de vida longo e feitos para os clientes, como aqueles que controlam máquinas complexas. Os clientes podem usar um único release desses sistemas por muitos anos e requerer mudanças específicas para determinado release de software muito tempo depois da sua data de liberação original.

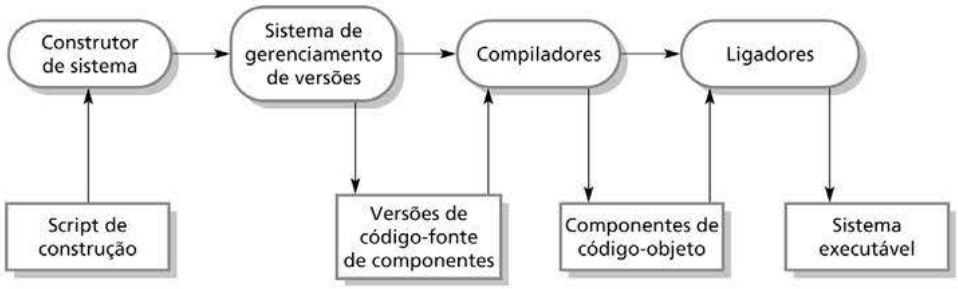
Para documentar um release, você deve registrar as versões específicas dos componentes de código-fonte usados para criar o código executável. Você deve manter cópias dos códigos-fonte e código executável, e de todos os arquivos de dados e de configuração. Você deve também registrar as versões do sistema operacional, as bibliotecas, os compiladores e outras ferramentas usadas para construir o software. Elas podem ser necessárias para construir exatamente o mesmo sistema em alguma data posterior. Isso pode significar que você deve armazenar cópias da plataforma de software e as ferramentas usadas para criar o sistema no sistema de gerenciamento de versão junto com o código-fonte do sistema-alvo.

29.4 Construção de sistemas

A construção de sistemas é um processo de compilação e ligação de componentes de software num programa que executa determinada configuração definida. Quando você está construindo um sistema com base em seus componentes, você deve pensar nas seguintes questões:

1. Todos os componentes que compõem um sistema foram incluídos nas instruções de construção?
2. A versão apropriada de cada componente necessário foi incluída nas instruções de construção?
3. Todos os arquivos de dados necessários estão disponíveis?
4. Se os arquivos de dados estão referenciados dentro de um componente, o nome usado é o mesmo que o do arquivo de dados na máquina-alvo?
5. A versão apropriada do compilador e de outras ferramentas requeridas está disponível? As versões atuais das ferramentas de software podem ser incompatíveis com as versões antigas usadas para desenvolver o sistema.

Figura 29.7
Construção de sistemas.



Hoje em dia, ferramentas de gerenciamento de configurações ou, algumas vezes, de ambiente de programação são usadas para automatizar o processo de construção de sistemas. A equipe de CM escreve um script de construção que define as dependências entre os componentes do sistema. O scritp também define as ferramentas usadas para compilar e ligar os componentes do sistema. As ferramentas de construção de sistema interpretam o script de construção e chamam outros programas, quando necessário, para construir um sistema executável baseado em seus componentes. Isso é ilustrado na Figura 29.7. Em alguns ambientes de programação (como ambiente de desenvolvimento Java), o script de construção é criado automaticamente pela análise do código-fonte e pela definição de quais componentes são chamados. Naturalmente, nessa situação, o nome do componente armazenado tem de ser igual ao nome do componente do programa.

As dependências entre componentes são especificadas no script de construção. Ele fornece informações, de modo que a ferramenta de construção do sistema possa decidir quando o código-fonte dos componentes deve ser recompilado e quando o código-objeto atual pode ser reusado. Em diversas ferramentas, essas dependências de script de construção são freqüentemente especificadas como dependências entre arquivos físicos nos quais o código-fonte e o código-objeto dos componentes são armazenados. Todavia, quando há múltiplos arquivos de códigos-fonte representando múltiplas versões de componentes, pode ser difícil dizer quais arquivos-fonte foram usados para derivar componentes de código-objeto. Essa confusão é particularmente a mesma quando a correspondência entre arquivos de código-fonte e objeto liberados possuem o mesmo nome, mas diferentes sufixos (isto é, .c e .o). Esse problema pode ser resolvido somente quando as ferramentas de gerenciamento de versões e de construção de sistemas estão integradas.

29.5 Ferramentas CASE para gerenciamento de configurações

Processos de gerenciamento de configurações são normalmente padronizados e envolvem aplicações de procedimentos predefinidos. Eles requerem o gerenciamento cuidadoso de grande quantidade de dados e é essencial a atenção aos detalhes. Quando um sistema está sendo construído com base em versões de componentes, um único erro de gerenciamento de configuração pode significar que o software não irá operar adequadamente. Conseqüentemente, o apoio de uma ferramenta CASE é essencial para o gerenciamento de configuração e, desde 1970, muitas ferramentas de software que abrangem diferentes áreas de gerenciamento de configuração foram produzidas.

Essas ferramentas podem ser combinadas para criar uma área de trabalho para apoiar todas as atividades de CM. Há dois tipos de workbench de CM:

1. *Workbenches abertos*. Ferramentas para cada estágio do processo CM são integradas por meio de procedimentos organizacionais padronizados para uso dessas ferramentas. Há muitas ferramentas comerciais e de código-fonte aberto disponíveis para propósitos específicos. O gerenciamento de mudanças pode ser apoiado pelas ferramentas para rastrear defeitos, como a Bugzilla, o gerenciamento de versões pelo uso de ferramentas, como RCS (Tichy, 1985) ou CVS (Berliner, 1990), e a construção de sistemas pelo uso de ferramentas, como a *make* (Feldman, 1979; Oram e Talbott, 1991) ou a *imake* (DuBois, 1996). Essas são todas as ferramentas de código-fonte aberto que estão disponíveis gratuitamente.
2. *Workbenches integrados.* Fornecem recursos integrados para controlar versões, a construção de sistemas e o rastreamento de mudanças. Por exemplo, o processo *Unified Change Management* da Rational conta com um workbench integrado de CM que incorpora o ClearCase (White, 2000) para a construção de sistemas e gerenciamento de versões e o ClearQuest para rastreamento das mudanças. As vantagens de um workbench integrado de CM são a troca de dados simplificada, e o workbench inclui um banco de dados integrado de CM. Workbenches integrados de SCM integrada foram derivados de alguns sistemas anteriores, como o Lifespan (Whitgift, 1991) para o gerenciamento de mudanças e o DSEE (Leblang e Chase, 1987) para o gerenciamento de versões e construção de sistemas. Entretanto, os workbenches integrados de CM são complexos e dispendiosos, e muitas organizações preferem usar ferramentas de apoio individuais, mais baratas e simples.

Muitos sistemas grandes são desenvolvidos em diferentes localidades e necessitam de ferramentas de SCM que apóiem o trabalho em múltiplas localidades com múltiplos repositórios de dados para itens de configuração. Enquanto a maioria das ferramentas de SCM é projetada para trabalhar num único local, algumas ferramentas, como CVS, têm recursos para dar suporte a múltiplas localidades (Vesperman, 2003).

29.5.1 Apoio para gerenciamento de mudanças

Cada pessoa envolvida no processo de gerenciamento de mudanças é responsável por alguma atividade. Elas completam essas atividades e passam os formulários e itens de configuração associados para os outros. O procedimento natural desse processo significa que um modelo de processo de mudanças pode ser projetado e integrado com um sistema de gerenciamento de versões. Esse modelo pode ser interpretado de modo que os documentos certos são passados para pessoas certas no tempo certo.

Há diversas ferramentas de gerenciamento de mudanças disponíveis, desde as mais simples, de código-fonte aberto, como o Bugzilla, até sistemas abrangentes integrados, como o ClearQuest da Rational. Essas ferramentas fornecem alguns ou todos os seguintes recursos para dar suporte ao processo:

1. *Um editor de formulários* que permite às pessoas criar ou completar formulários com propostas de mudanças.
2. *Um sistema de workflow* que permite à equipe de CM definir quem deve processar o formulário de solicitação de mudança e a ordem de processamento. Esse sistema também encaminhará automaticamente os formulários para as pessoas corretas no tempo certo e informará aos membros relevantes da equipe sobre o progresso da mudança. O e-mail é usado para prover atualizações de progresso para todos aqueles envolvidos no processo.
3. *Um banco de dados de mudança* que é usado para gerenciar todas as propostas de mudança e que pode estar ligado a um sistema de gerenciamento de versões. Recursos de consulta no banco de dados permitem à equipe de CM encontrar uma proposta específica de mudança.
4. *Um sistema de relato de mudanças* que gera relatórios gerenciais sobre o status das solicitações de mudança enviadas.

29.5.2 Apoio para gerenciamento de versões

O gerenciamento de versões envolve o gerenciamento de grande quantidade de informação e assegura que as mudanças de sistema sejam registradas e controladas. Ferramentas de gerenciamento de versões controlam um repositório de itens de configuração no qual os conteúdos dos repositórios são imutáveis (isto é, não podem ser alterados). Para trabalhar num item de configuração, você deve retirá-lo (check-out) do repositório e guardá-lo num diretório de trabalho. Depois que realizou as mudanças no software, deve-se devolvê-lo (check-in) ao repositório, e uma nova versão é automaticamente criada.

Todos os sistemas de gerenciamento de versões fornecem um conjunto básico de capacidades semelhantes, embora alguns recursos sejam mais sofisticados que outros. Exemplos dessas capacidades são:

1. *Identificação de versões e releases.* Versões gerenciadas recebem identificadores quando são enviadas ao sistema. Sistemas diferentes apóiam tipos distintos de identificação de versões, explicados na Seção 29.3.1.
2. *Gerenciamento de armazenamento.* Para reduzir o espaço de armazenamento exigido pelas várias versões, que são em grande parte as mesmas, um sistema de gerenciamento de versões fornece recursos de gerenciamento de armazenamento de maneira que as versões sejam descritas pela sua diferença em relação a uma versão mestre. Diferenças entre versões são representadas por um *delta*, que engloba as instruções necessárias para recriar a versão do sistema associado. Isso é ilustrado na Figura 29.8, a qual mostra como os deltas anteriores podem ser aplicados na última versão de um sistema para recriar versões de sistema mais recentes. A última versão é a versão 1.3. Para criar a versão 1.2, você aplica o delta de mudança que recria essa versão.
3. *Registro do histórico de mudanças.* Todas as mudanças executadas no código de um sistema ou no componente são registradas e listadas. Em alguns sistemas, essas mudanças devem ser usadas para selecionar uma versão específica do sistema.
4. *Desenvolvimento independente.* Múltiplas versões de um sistema podem ser desenvolvidas em paralelo e cada versão pode ser modificada independentemente. Por exemplo, o release 1 pode ser modificado depois que o desenvolvimento do release 2 estiver em andamento mediante a adição de deltas novos de nível 1. O sistema de gerenciamento de versões mantém a rastreabilidade dos componentes retirados (check-out) para a edição e assegura que as mudanças efetuadas no mesmo componente pelos diferentes desenvolvedores não interfiram. Alguns sistemas permitem somente

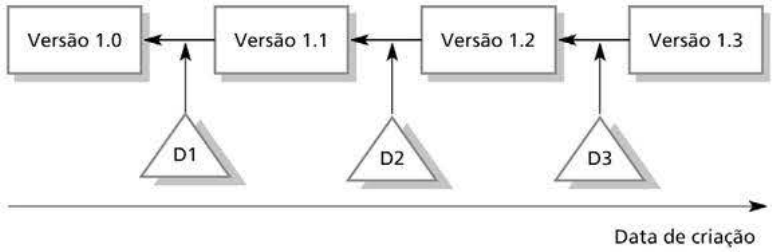


Figura 29.8
Versões baseadas em delta.

a retirada de uma instância de componente para a edição, outros resolvem potenciais batimentos quando os componentes editados são devolvidos (check-in) ao sistema.

5. *Suporte a projetos.* O sistema pode dar suporte a múltiplos projetos tão bem quanto a múltiplos arquivos. Em sistemas de apoio a projetos, como o CVS, é possível retirar e devolver todos os arquivos associados com um projeto antes de trabalhar apenas com um arquivo num determinado tempo.

29.5.3 Suporte para construção de sistemas

A construção de sistemas é um processo computacional intensivo. A compilação e a ligação de todos os componentes de um sistema grande podem levar muitas horas. Pode haver centenas de arquivos envolvidos e, conseqüentemente, a possibilidade de erro humano se eles forem compilados e ligados manualmente. As ferramentas de construção de sistemas automatizam o processo de construção para reduzir o potencial de erro humano e, quando possível, minimizar o tempo necessário para a construção de sistemas.

Ferramentas de construção de sistemas podem ser dedicadas, como derivadas das utilidades do Unix (Oram e Talbott, 1991), ou podem ser integradas com ferramentas de gerenciamento de versões. Recursos oriundos de ferramentas CASE de construção de sistemas podem incluir:

1. *Uma linguagem de especificação de dependência e um interpretador associado.* Dependências de componentes podem ser descritas e a recompilação, minimizada. Esclareço isso detalhadamente mais adiante nesta seção.
2. *Seleção de ferramentas e apoio à instanciação.* Os compiladores e outras ferramentas de processamento usadas para processar os arquivos de código-fonte podem ser especificados e instanciados quando necessário.
3. *Compilação distribuída.* Alguns construtores de sistemas, especialmente os que são parte de sistemas integrados de CM, suportam a compilação distribuída em rede. Antes de todas as compilações serem executadas em uma única máquina, o construtor de sistema procura por um processador ocioso na rede e aloca um número de compilações paralelas. Isso reduz significativamente o tempo necessário para construir um sistema.
4. *Gerenciamento de objetos derivados.* Os objetos derivados são objetos criados de outros objetos-fonte. O gerenciamento de objetos derivados liga-os ao código-fonte e deriva novamente um objeto somente quando ele for requerido pelas mudanças do código-fonte.

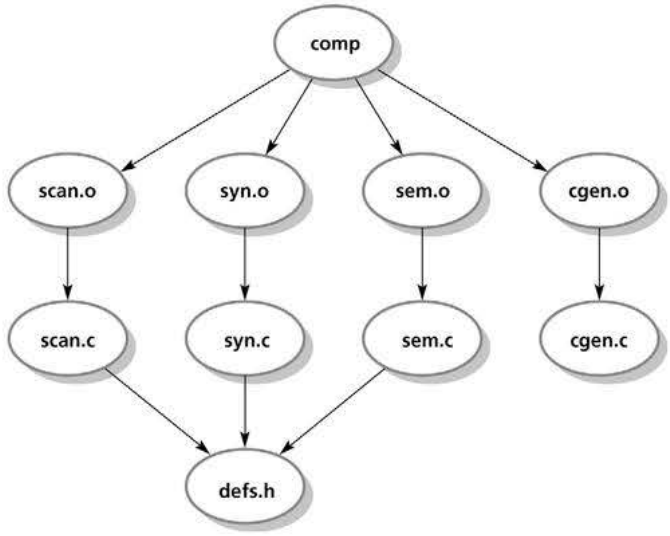
O gerenciamento de objetos derivados e a minimização da recompilação são mais bem explicados com a utilização de um exemplo simples. Considere uma situação em que um programa de compilador chamado **comp** é criado por quatro módulos objetos chamados **scan.o**, **syn.o**, **sem.o** e **cgen.o**. Cada módulo objeto é criado do módulo de código-fonte com os correspondentes nomes (**scan.c**, **syn.c**, **sem.c** e **cgen.c**). Um arquivo de declarações de variáveis e de constantes chamado defs.h é compartilhado por **scan.c**, **syn.c** e **sem.c** (Figura 29.9). Na Figura 29.9, as setas significam 'depende de' — a entidade na base da seta depende da entidade que está no topo. Portanto, **comp** depende de **scan.o**, **syn.o**, **sem.o** e **cgen.o**; **scan.o** depende de **scan.c** e assim por diante.

Se **scan.c** for alterado, a ferramenta de construção de sistemas pode detectar que o objeto derivado scan.o deve ser recriado. Ela faz isso pela comparação dos momentos da modificação do **scan.o** e do **scan.c**, e detecta que **scan.c** foi modificado depois de **scan.o**. Depois ela chama o compilador C para compilar o **scan.c** a fim de criar um novo objeto derivado, **scan.o**.

A ferramenta de construção então usa a ligação de dependência entre comp e scan.o para detectar que **comp** deve ser recriado pela ligação de **scan.o**, **syn.o**, **sem.o** e **cgen.o**. O sistema pode detectar que outros componentes de código objeto estão inalterados; assim, a recompilação de seu código-fonte não é necessária.

Muitas ferramentas de construção de sistemas usam a data de modificação de arquivo como um atributo-chave para decidir se a recompilação é necessária. Se um arquivo de código-fonte é modificado depois do seu correspondente arquivo de código-objeto, o arquivo de código objeto é recriado. Essencialmente, pode haver somente uma versão do código-objeto

Figura 29.9
Dependências entre componentes.



que corresponde ao mais recente componente de código-fonte modificado. Quando uma nova versão de componente de código-fonte é recriada, perde-se o código-objeto da versão anterior.

No entanto, algumas ferramentas usam abordagens mais sofisticadas para o gerenciamento de objetos derivados. Elas colocam uma marca nos objetos derivados com a identificação do código-fonte usado para gerar os objetos. Dentro dos limites de capacidade de armazenamento, elas mantêm todos os objetos derivados. Portanto, geralmente é possível recuperar o código objeto de todas as versões de componentes de códigos-fonte sem recompilação.

PONTOS-CHAVE

- O gerenciamento de configuração é o gerenciamento das mudanças do sistema. Quando um sistema passa por manutenção, o papel da equipe de CM é assegurar que as mudanças sejam incorporadas de maneira controlada.
- Em projetos grandes, um esquema formal de atribuição de nomes deve ser definido e usado como base para manter a rastreabilidade das versões de todos os documentos do projeto.
- A equipe de CM deve ser apoiada por um banco de dados de configuração que registra informações sobre as mudanças do sistema e as solicitações de mudanças pendentes. Os projetos devem ter algum meio formal de solicitação de mudanças de sistema.
- Na definição de um esquema de gerenciamento de configurações, um esquema consistente de identificação de versões deve ser definido. As versões podem ser identificadas pelo número da versão, por um conjunto de atributos associados ou pelas mudanças de sistema propostas que elas implementam.
- Os sistemas liberados incluem o código executável, os arquivos de dados, os arquivos de configuração e a documentação. O gerenciamento de releases envolve a tomada de decisão sobre a data de liberação de sistema, o preparo de todas as informações para distribuição e a documentação de cada release de sistema.
- A construção de sistema é o processo de montagem de componentes do sistema em um programa executável para rodar no sistema computacional-alvo.
- As ferramentas CASE estão disponíveis para apoiar todas as atividades do gerenciamento de configuração. Elas incluem ferramentas como CVS para gerenciar as versões de sistema, ferramentas para apoiar o gerenciamento das mudanças e ferramentas para a construção do sistema.
- As ferramentas CASE para CM podem ser dedicadas para apoiar o gerenciamento de mudanças e de versões e construção de sistemas, ou podem ser workbenches integrados que fornecem uma única interface para todo apoio de CM.

LEITURAS SUGERIDAS

Configuration management principle and practice. Este livro, muito abrangente, apresenta abordagens padrão e tradicionais para CM como também abordagens de CM mais adequadas a processos modernos, como processos ágeis de desenvolvimento de software (A. M. J. Hass, 2002, Addison-Wesley).

"A layered architecture for uniform version management". Este artigo explica as diferentes abordagens para gerenciamento das versões de software e propõe um modelo básico que possa acomodar todos eles. Inclui, particularmente, um bom levantamento sobre a experiência de trabalho nessa área (B. Westfechel et al., *IEEE Transactions on Software Engineering*, 27 (12), dezembro de 2001).

"Software configuration management: a roadmap". Este artigo oferece uma visão geral da evolução do SCM e identifica desafios de pesquisa que permanecem nessa área (J. Estublier, *Proc. Int. Conf. on Software Engeneering*, 2000. IEEE Press).

Trends in software: configuration management. Trata-se de um conjunto de artigos sobre aspectos de gerenciamento de configurações escrito por autores que são pesquisadores e praticantes ativos nessa área. É uma boa fonte de informações para estudantes e praticantes interessados em tópicos avançados de CM. A maior parte desses problemas ainda não foi resolvida (W. Tichy (ed.), 1995, John Wiley & Sons).

EXERCÍCIOS

29.1 Explique por que não se deve usar o título de um documento para identificá-lo no sistema de gerenciamento de configuração. Sugira um padrão para um esquema de identificação de documentos que possa ser usado por todos os projetos em uma organização.

29.2 Usando uma abordagem orientada a objetos (veja o Capítulo 8), projete um modelo de banco de dados de configuração que registra informações sobre os componentes de sistemas, releases e mudanças. Alguns requisitos para o modelo de dados são os seguintes:

- Deve ser possível recuperar todas as versões ou uma simples versão identificada de um componente.
- Deve ser possível recuperar a última versão de um componente.
- Deve ser possível encontrar quais solicitações de mudança foram implementadas para uma versão específica de um sistema.
- Deve ser possível descobrir quais versões de componentes estão incluídas em uma versão específica de um sistema.
- Deve ser possível recuperar um release particular de um sistema de acordo tanto com a data de sua produção quanto por meio dos clientes aos quais o release foi entregue.

29.3 Usando um diagrama de fluxo de dados, descreva um procedimento de gerenciamento de mudanças que pode ser usado em uma grande organização envolvida com o desenvolvimento de software para clientes externos. As mudanças podem ser sugeridas tanto de fontes externas quanto de fontes internas.

29.4 Como o uso de um sistema de gerenciamento de configuração baseado em projeto, como o CVS, simplifica o processo de gerenciamento de versões?

29.5 Explique por que um sistema de identificação de versão baseado em atributos facilita a descoberta de todos os componentes que constituem uma versão específica de um sistema.

29.6 Descreva as dificuldades que podem surgir quando construímos um sistema com base em seus componentes. Quais problemas específicos podem ocorrer quando um sistema é construído em um computador hospedeiro para alguma máquina-alvo?

29.7 Com referência à construção de sistemas, explique por que você algumas vezes pode precisar manter computadores obsoletos a partir dos quais sistemas grandes de software foram desenvolvidos.

29.8 Um problema comum com a construção de sistemas ocorre quando nomes de arquivos físicos são incorporados ao código do sistema e a estrutura de arquivos envolvida com esses nomes difere dos da máquina-alvo. Escreva um conjunto de diretrizes do programador para ajudar a evitar esse problema e outros problemas de construção de sistemas sugeridos por você.

29.9 Descreva cinco fatores que devem ser considerados pelos engenheiros durante o processo de construção de uma versão de um sistema de software de grande porte.

29.10 Descreva duas maneiras pelas quais as ferramentas de construção de sistemas podem otimizar o processo de construção de uma versão de um sistema com base em seus componentes.

PARTE 7 TECNOLOGIAS EMERGENTES

Enquanto os fundamentos da engenharia de software são bastante estáveis, as tecnologias que apóiam o processo de engenharia de software estão sujeitas a mudanças contínuas. Existem diversas abordagens diferentes desenvolvidas em laboratórios de pesquisas que estão começando a passar ao uso prático. Nesta seção, explico três abordagens que considero particularmente importantes. Acredito que elas possam influenciar profundamente a engenharia de software no futuro.

Essas três abordagens estão em diferentes estágios de desenvolvimento. A engenharia de proteção, tópico do Capítulo 30, já está sendo usada no desenvolvimento de sistemas reais. A engenharia de software orientada a serviços ainda está em um estágio inicial de desenvolvimento, mas há um investimento incrível nessa abordagem por parte de todas as grandes empresas. Ela, portanto, certamente se tornará uma importante tecnologia de desenvolvimento de software em um futuro próximo. O desenvolvimento de software orientado a aspectos está menos maduro, mas inclui idéias incríveis que poderão ter grande impacto na maneira como o software é organizado.

Os capítulos desta parte do livro são:

1. O Capítulo 30 está relacionado à engenharia de proteção. Incluí esse capítulo devido, infelizmente, ao fato de que a ameaça a sistemas por ataques maliciosos de criminosos e vândalos vem crescendo cada vez mais. O capítulo aborda o material sobre engenharia de sistemas críticos, que é um tema tratado no restante do livro. Neste capítulo explico conceitos fundamentais de proteção, uma abordagem dirigida a riscos para derivar requisitos de proteção. Sugiro um guia de projeto para sistemas de proteção e introduzo a importante noção de projeto para a capacidade de sobrevivência de sistemas.
2. O tema do Capítulo 31 é a engenharia de software orientada a serviços, ampliando a explicação de arquiteturas orientadas a serviços do Capítulo 12. Apresento o conceito de serviço como um componente reusável e explico o processo de projeto e implementação de serviços (engenharia de serviços). Depois explico como o software pode ser desenvolvido usando serviços como blocos fundamentais de construção.
3. O Capítulo 32 aborda o desenvolvimento de software orientado a aspectos. Esta é uma nova abordagem para o desenvolvimento de software, que enfoca explicitamente o apoio à separação de assuntos. Como essa abordagem ainda é experimental, o foco se fixa na introdução de conceitos e a descrição de como uma perspectiva orientada a aspectos pode ser adotada em todos os estágios do processo de software. Explico também uma importante barreira para a ampla adoção do desenvolvimento orientado a aspectos, a saber, as dificuldades de testar sistematicamente programas orientados a aspectos.

30

Engenharia de proteção

Objetivos

O objetivo deste capítulo é apresentar questões que precisam ser consideradas na especificação e no projeto de software seguro. Após ler este capitulo, você:

- compreenderá a importância do gerenciamento de riscos de proteção e como os requisitos de proteção podem ser derivados de uma análise de riscos;
- saberá como considerações de proteção devem influenciar o projeto de arquiteturas de sistemas e boas práticas de projeto para o desenvolvimento de sistemas seguros;
- compreenderá a noção da capacidade de sobrevivência de sistemas e o motivo da análise de capacidade de sobrevivência ser importante para sistemas de software complexos.

Conteúdo

30.1 Conceitos de proteção
30.2 Gerenciamento de riscos de proteção
30.3 Projeto para proteção
30.4 Capacidade de sobrevivência de sistemas

A difusão do uso da Internet na década de 1990 introduziu um novo desafio para os engenheiros de software: projetar e implementar sistemas seguros. Como mais e mais sistemas foram conectados à Internet, uma variedade de diferentes ataques externos foi inventada, e eles ameaçam esses sistemas. Os problemas de produção de sistemas confiáveis aumentaram drasticamente. Os engenheiros de sistemas tiveram que considerar ameaças de agressores maliciosos e tecnicamente experientes, bem como problemas resultantes de erros acidentais em processos de desenvolvimento.

Atualmente, é essencial projetar sistemas para se defender dos ataques externos e se recuperar desses ataques. Sem precauções de proteção, é quase inevitável que os agressores comprometam os sistemas em rede. Eles podem fazer um uso errôneo do sistema de hardware, roubar dados confidenciais ou interromper os serviços oferecidos pelo sistema. A engenharia de proteção de sistema é, conseqüentemente, um aspecto importante em crescimento no processo de engenharia de sistema.

A engenharia da proteção está relacionada ao desenvolvimento e à manutenção dos sistemas que podem resistir aos ataques maliciosos, com a intenção de danificar um sistema baseado em computador ou seus dados. A engenharia da proteção é parte do campo mais geral da proteção de computador. Isso se transformou em uma prioridade para os negócios e para os indivíduos,

quando mais e mais criminosos tentam explorar os sistemas em rede para finalidades ilegais. Os engenheiros de software devem estar cientes das ameaças de proteção enfrentadas pelos sistemas e da maneira como essas ameaças podem ser neutralizadas.

Minha intenção neste capítulo é apresentar a engenharia de proteção aos engenheiros de software, com foco na especificação e no projeto de proteção de aplicações. Apresento conceitos e princípios que apóiam a engenharia de software de sistemas seguros. Não tento resumir a proteção de computador como um todo e, assim, não abordo tópicos tais como mecanismos de criptografia, controle de acesso e autenticação. Eles estão descritos em detalhes em textos mais gerais sobre proteção de computador (Anderson, 2001; Pfleeger e Pfleeger, 2003; Bishop, 2005).

Este capítulo inclui a explicação de tópicos de proteção tratados em outros capítulos. Você deve ler o material deste capítulo em conjunto com:

- Seção 3.5, que explica a proteção como propriedade de confiança;
- Seção 9.3, que explica a especificação de proteção;
- Seção 24.3, que explica a avaliação de proteção.

Ao tratar de questões de proteção, você precisa considerar tanto o software de aplicação (o sistema de controle, o sistema de informações etc.) quanto a infra-estrutura com a qual esse sistema é construído (Figura 30.1). A infra-estrutura para aplicações complexas inclui uma plataforma de sistema operacional, tal como Linux ou Windows; outras aplicações genéricas que rodam nesse sistema, tais como clientes com navegadores Web e e-mail; um sistema de gerenciamento de banco de dados; middleware que apóie a computação distribuída e o acesso ao banco de dados e bibliotecas de componentes reusáveis empregados pelo software de aplicação. De fato, a maioria dos ataques foca as infra-estruturas de sistema, uma vez que os componentes (por exemplo, navegadores Web) são bem conhecidos e amplamente disponíveis.

Na prática, há uma importante distinção entre a proteção de aplicações e a proteção de infra-estrutura:

1. A proteção de aplicações é um problema de engenharia de software. Os engenheiros de software devem assegurar que o sistema está *projetado* para resistir aos ataques.
2. A proteção de infra-estrutura é um problema do gerenciamento de sistemas. Os gerentes de sistema devem assegurar que a infra-estrutura está *configurada* para resistir aos ataques. Eles precisam definir a infra-estrutura para fazer uso mais eficiente de quaisquer características de proteção de infra-estrutura disponibilizadas. Eles também precisam corrigir as vulnerabilidades de proteção de infra-estrutura que surgem quando o software é usado.

Gerenciamento de proteção não é uma tarefa simples, mas inclui uma série de atividades tais como gerenciamento de usuários e permissões, implantação e manutenção de software de sistema e monitoração, detecção e recuperação de ataques:

1. O gerenciamento de usuários e permissões inclui adicionar e remover usuários do sistema, assegurar que os mecanismos adequados de autenticação estejam funcionando e estabelecer as permissões no sistema de modo que os usuários tenham acesso somente aos recursos de que necessitam.
2. A implantação e manutenção de sistema de software incluem a instalação de sistema de software e de middleware e sua configuração adequada para que as vulnerabilidades de proteção sejam evitadas. Isso também envolve a atualização regular desse software com as novas versões ou patches que reparam os problemas de proteção descobertos.
3. A monitoração, detecção e recuperação de ataques incluem atividades que monitoram o sistema em relação a acessos não autorizados, detectam e colocam em funcionamento as estratégias para resistir aos ataques e atividades de backup de modo que as operações normais possam ser reassumidas depois de um ataque externo.

Figura 30.1
Camadas de sistema onde a segurança pode ser comprometida.

O gerenciamento de proteção é de vital importância para a manutenção de sistemas de proteção. No entanto, as questões principais dos engenheiros de software não são os processos de gerenciamento, mas como projetar suporte ao gerenciamento de proteção nos seus sistemas de aplicação. Explico o projeto para gerenciamento de sistemas na Seção 30.3.3.

30.1 Conceitos de proteção

No Capítulo 3, apresentei alguns conceitos e terminologias básicas de proteção. A Tabela 30.1, pequena versão modificada da Figura 3.9, é um lembrete de alguns desses conceitos básicos. Para auxiliar a compreensão dos conceitos, considere o seguinte cenário.

Um sistema de informações hospitalares mantém as informações pessoais sobre seus pacientes com problemas de saúde mental e seus tratamentos. Como questões de saúde mental são particularmente sensíveis, é essencial que a privacidade dos pacientes seja mantida. Esse sistema precisa ser acessível de hospitais e clínicas diferentes, de modo que foi estabelecido que os usuários acessem o sistema por meio de um navegador Web. O pessoal do hospital deve fazer o logon no sistema usando um nome de usuário e senha. O sistema exige que a senha tenha pelo menos oito letras, mas permite que qualquer senha seja fornecida sem verificação posterior.

Um criminoso foi informado de que um astro do esporte bem pago está fazendo tratamento referente a problemas de saúde mental. Ele gostaria de obter acesso ilegal às informações no sistema, de modo que possa fazer chantagem com o astro. Assumindo ser um parente preocupado e conversando com as enfermeiras de uma clínica de saúde mental, ele descobre como acessar o sistema. Verificando os nomes nos crachás, identifica os nomes de algumas pessoas que possuem permissão de acesso. Ele então tenta fazer o logon no sistema usando esses nomes e adivinhando sistematicamente senhas possíveis.

A Tabela 30.2 mostra os conceitos de proteção descritos na Tabela 30.1 e como eles se relacionam com esse cenário do sistema de informação hospitalar.

As ameaças de proteção enquadram-se em três categorias principais:

1. Ameaças à *confidencialidade* do sistema e a seus dados. Essas ameaças podem revelar informações a pessoas ou programas que não estão autorizados a ter acesso a elas.
2. Ameaças à *integridade* do sistema e a seus dados. Essas ameaças podem danificar ou corromper o software ou seus dados.
3. Ameaças à *disponibilidade* do sistema e a seus dados. Essas ameaças podem restringir acessos ao software ou a seus dados para usuários autorizados.

As ameaças são, naturalmente, interdependentes. Se um ataque torna o sistema indisponível, você não será capaz de atualizar as informações que mudam com o tempo. Isso significa que a integridade do sistema pode ser comprometida. Se um ataque for bem-sucedido e a integridade do software, comprometida, pode ser necessário desligar o sistema para reparar o problema. Portanto, a disponibilidade do sistema é reduzida.

Tabela 30.1 Conceitos de proteção

Termo	Descrição
Ativo	Recurso de sistema que possui um valor e deve ser protegido.
Exposição	Possível perda ou dano que pode resultar de um ataque bem-sucedido. Pode ser uma perda ou dano em dados ou uma perda de tempo e esforço, caso seja necessária uma recuperação depois de uma brecha de proteção.
Vulnerabilidade	Ponto fraco em um sistema baseado em computadores que pode ser explorado para causar perda ou dano.
Ataque	Exploração de uma vulnerabilidade do sistema. Geralmente, é externo ao sistema e constitui uma tentativa deliberada de causar algum dano.
Ameaça	Circunstâncias com potencial para causar perda ou dano. Você pode pensar nelas como uma vulnerabilidade de sistema sujeita a ataque.
Controle	Medida de proteção que reduz uma vulnerabilidade de sistema. A criptografia pode ser um exemplo de controle que reduz uma vulnerabilidade de um sistema fraco em controle de acesso.

Tabela 30.2 Exemplos de conceitos de proteção

Termo	Descrição
Ativo	Os registros de cada paciente que está recebendo ou recebeu tratamento.
Exposição	Prejuízo financeiro potencial devido a futuros pacientes que não procurarão tratamento por não confiarem na clínica para manter seus dados. Prejuízo financeiro devido a ação jurídica movida pelo astro do esporte. Perda de reputação.
Vulnerabilidade	Sistema fraco em senhas que permite aos usuários criarem senhas que podem ser descobertas. IDs de usuário iguais a seus nomes.
Ataque	A imitação de um usuário autorizado.
Ameaça	Um usuário não autorizado ganhará acesso ao sistema descobrindo as credenciais (nome de login e senha) de um usuário autorizado.
Controle	Um sistema de verificação de senhas que desautoriza senhas definidas pelos usuários que sejam nomes próprios e palavras normalmente incluídas em um dicionário.

Os controles que você poderia colocar em funcionamento para aumentar a proteção de sistema também se enquadram em três classes:

1. Controles que se destinam a assegurar que os ataques sejam malsucedidos. A estratégia aqui é projetar o sistema de modo que os problemas de proteção sejam evitados. Por exemplo, sistemas militares sensíveis não são conectados à rede pública, de modo que o acesso externo é impossível. Você deve também pensar na criptografia com um controle baseado em prevenção. Qualquer acesso não autorizado aos dados criptografados significa que eles não poderão ser lidos pelo agressor. Na prática, é muito oneroso e consome muito tempo quebrar uma criptografia bem protegida.
2. Controles que se destinam a detectar e repelir ataques. Esses controles envolvem a inclusão de funcionalidade em um sistema que monitora a sua operação e verifica padrões não usuais de atividade. Se eles são detectados, a ação pode ser tomada, como desligar partes do sistema, restringindo o acesso a certos usuários etc.
3. Controles que apóiam a recuperação de problemas. Esses controles podem abranger estratégias automáticas de back-up e 'espelhamento' de informações por meio de políticas de seguro que cobrem os custos associados com ataques bem-sucedidos ao sistema.

A engenharia de proteção é parte de um processo mais geral da engenharia de confiança. Portanto, tem muito mais em comum com a engenharia de segurança e de confiabilidade. Estratégias básicas contam com prevenção, detecção de problemas e recuperação que ocorreram. Bons sistemas não contam com uma única estratégia, mas usam uma combinação sensata de todas elas para alcançar os objetivos de proteção.

30.2 Gerenciamento de riscos de proteção

O gerenciamento e a avaliação de riscos de proteção são essenciais para uma engenharia de proteção eficiente. O gerenciamento de riscos está relacionado à avaliação de possíveis perdas que poderiam resultar de ataques em ativos do sistema e ao balanço dessas perdas em relação aos custos de procedimentos de proteção que podem reduzi-las. As empresas de cartão de crédito fazem isso o tempo todo. É relativamente fácil introduzir novas tecnologias para reduzir fraudes em cartões de crédito, mas o custo disso seria maior do que a cobertura de perdas dos usuários de cartão de crédito. À medida que os custos caem e os ataques aumentam, esse equilíbrio pode mudar. Por exemplo, empresas de cartão de crédito estão atualmente codificando informações em um chip no cartão, no lugar de uma trilha magnética. Isso torna a cópia do cartão muito mais difícil.

O gerenciamento de riscos é uma questão de negócio, e não técnica, portanto os engenheiros de software não devem decidir quais controles seriam incluídos no sistema. É da gerência sênior a decisão de aceitar ou não o custo de proteção ou a exposição que resulta da falta de procedimentos de proteção. No entanto, o papel dos engenheiros de software é fornecer um guia técnico e julgamentos sobre questões de proteção. Eles são, portanto, participantes essenciais no processo de gerenciamento de riscos.

Uma entrada crítica para o processo de avaliação e gerenciamento de riscos é a política de proteção organizacional. Tal política se aplica a todos os sistemas e deve estabelecer o que pode ou não ser permitido. Por exemplo, um aspecto da política

de proteção militar pode estabelecer 'Os leitores podem somente examinar documentos cuja classificação seja a mesma ou inferior ao nível atribuído ao leitor'. Isso significa que se um leitor foi avaliado para o nível 'secreto', ele pode acessar documentos classificados como 'secreto', 'confidencial' ou 'aberto', mas não documentos classificados como 'ultra-secreto'.

A política de proteção estabelece condições que deveriam ser mantidas por um sistema de proteção e, assim, auxilia na identificação das ameaças que poderiam surgir. Risco é qualquer coisa que poderia ameaçar a proteção do negócio. Em princípio, como explicado por Bishop (2005), as políticas de proteção podem ser estabelecidas formalmente e várias verificações automáticas são feitas em relação a elas. Na prática, elas são normalmente documentos informais que definem o que é e o que não é permitido.

A avaliação de riscos inicia-se antes de a decisão de adquirir o sistema ter sido feita e deve continuar durante todo o processo de desenvolvimento do sistema. Uma importante consideração é a quantidade de informação que você tem disponível sobre o sistema; assim, a avaliação de riscos é um processo por estágios:

1. *Avaliação de riscos preliminar.* Neste estágio, decisões sobre os requisitos detalhados de sistema, o projeto de sistema ou a tecnologia de implementação ainda não foram feitas. O objetivo desse processo de avaliação é, inicialmente, avaliar se os benefícios de desenvolvimento do sistema justificam ou não os riscos associados e, então, derivar os requisitos específicos de proteção para o sistema ser implementado. Você não tem informações sobre as vulnerabilidades potenciais do sistema ou os controles incluídos nos componentes de sistema reusados ou o middleware.
2. *Avaliação de riscos de ciclo de vida.* Esta avaliação acontece durante o ciclo de vida de desenvolvimento de sistema e é informada pelas decisões técnicas de projeto e implementação do sistema. Ela informa o processo de engenharia de requisitos de proteção. As vulnerabilidades conhecidas e potenciais são identificadas e esse conhecimento é usado para informar a tomada de decisão sobre a funcionalidade de sistema e como deve ser implementada, testada e implantada.

Enquanto os detalhes desses processos variam radicalmente de uma organização para outra, existem atividades fundamentais comuns a todos eles. Explico essas atividades nas duas seções seguintes.

30.2.1 Avaliação de riscos preliminar

O objetivo da avaliação de riscos preliminar é derivar requisitos de proteção para todo o sistema, não somente o software. Esses requisitos influenciam a escolha da plataforma de sistema e do middleware e servem como base para o desenvolvimento mais detalhado dos requisitos funcionais de software.

Os estágios essenciais da avaliação de riscos preliminar (Figura 30.2) são:

1. A identificação de ativos, na qual os ativos de sistema que podem exigir proteção são identificados. O sistema em si ou as funções de sistema específicas podem ser identificadas como ativos, bem como os dados associados com o sistema.
2. A avaliação de valores de ativos, na qual você estima o valor dos ativos identificado.
3. A avaliação de exposição, na qual você avalia as perdas potenciais associadas com cada ativo.
4. A identificação de ameaças, na qual você identifica as ameaças aos ativos de sistema.
5. A avaliação de probabilidade, na qual você estima a probabilidade de cada ameaça.
6. A identificação de controle, na qual você propõe os controles que poderiam ser colocados para proteger um ativo.
7. A avaliação de viabilidade, na qual você avalia a viabilidade técnica e os custos de controles propostos.
8. A definição de requisitos de proteção, na qual as avaliações de exposição, ameaças e controle são usadas para derivar um conjunto de requisitos de proteção de sistema. Podem ser requisitos para a infra-estrutura de sistema ou o sistema de aplicação.

Para ilustrar esse processo, considere o sistema de informação hospitalar para tratamento de saúde mental que apresentei anteriormente neste capítulo. Não tenho espaço para explicar uma avaliação de riscos completa nesta seção, mas uso esse sistema como fonte de exemplos. Mostrei esses exemplos como um fragmento de um relatório (tabelas 30.3 e 30.4) que poderia ser gerado do processo de avaliação de riscos preliminar. Esse relatório de análise de riscos preliminar é uma entrada para o processo de definição de requisitos de proteção.

Alguns exemplos dos requisitos de proteção que poderiam ser derivados da análise de riscos para o sistema de informação hospitalar são:

1. As informações de pacientes devem ser baixadas, no início da sessão clínica, do banco de dados para uma área segura do sistema-cliente usada pelo pessoal clínico.
2. As informações de pacientes não devem ser mantidas no sistema cliente depois que uma sessão clínica terminou.

Figura 30.2
Avaliação de riscos preliminar.

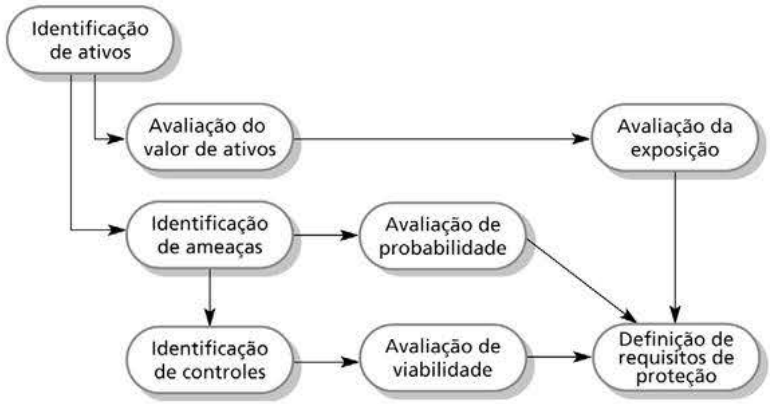


Tabela 30.3 Análise de ativos em um relatório de avaliação de riscos preliminar

Ativo	Valor	Exposição
Sistema de informações	Alto. Necessário para dar suporte a todas as consultas clínicas. Potencialmente crítico em segurança.	Alta. Prejuízo financeiro, na medida em que as consultas podem ser canceladas. Custos de restauração de sistema. Possível dano ao paciente se o tratamento não puder ser prescrito.
Banco de dados de pacientes	Alto. Necessário para apoiar todas as consultas clínicas. Potencialmente crítico em segurança.	Alta. Prejuízo financeiro, na medida em que as consultas podem ser canceladas. Custos de restauração de sistema. Possível dano ao paciente se o tratamento não puder ser prescrito.
Registro individual de paciente	Normalmente baixo, embora possa ser alto para pacientes específicos de perfil alto.	Baixos prejuízos diretos, mas possível perda de reputação.

Tabela 30.4 Análise de ameaças e controles em um relatório de avaliação de riscos preliminar

Ameaça	Probabilidade	Controle	Viabilidade
Usuário não autorizado ganha acesso como gerente de sistema e torna o sistema indisponível	Baixa	Permitir somente o gerenciamento do sistema com base em localizações específicas fisicamente protegidas.	Baixo custo de implementação, mas devem ser tomados cuidados com a distribuição de chaves e assegurar que estas estejam disponíveis no caso de uma emergência.
Um usuário não autorizado obtém acesso como usuário de sistema e acessa informações confidenciais	Alta	Requerer que todos os usuários se autentiquem usando um mecanismo biométrico. Fazer o log de todas as mudanças de informações de pacientes para acompanhar o uso do sistema.	Tecnicamente viável, mas uma solução de alto custo. Possível resistência do usuário. Simples e transparente de implementar e também dá suporte para a recuperação.

3. Um log deve ser mantido em um computador separado do servidor de banco de dados de todas as mudanças efetuadas no banco de dados de sistema.

Os dois primeiros requisitos estão relacionados — informações de paciente são baixadas para uma máquina local, de modo que as consultas podem continuar se o servidor de banco de dados do paciente for atacado. Por outro lado, essas informações devem ser excluídas, de modo que os usuários seguintes do computador cliente não possam acessá-las. O terceiro requisito é um requisito de recuperação e auditoria. Quer dizer que mudanças podem ser recuperadas examinando-se o log delas e, adicionalmente, rastreando quem as fez.

30.2.2 Avaliação de riscos de ciclo de vida

A distinção importante entre avaliação de riscos de ciclo de vida e avaliação de riscos preliminar é que, na avaliação de riscos de ciclo de vida, o conhecimento da arquitetura de sistema e da organização de dados está disponível. As decisões de aquisição foram feitas de modo que a plataforma de sistema e o middleware foram escolhidos. Uma estratégia de desenvolvimento, como 'configurar uma aplicação genérica', pode ter sido escolhida.

Isso significa que você tem muito mais informação detalhada sobre o que precisa ser protegido e saberá algo sobre as vulnerabilidades do sistema. Algumas dessas vulnerabilidades serão inerentes às escolhas de projeto feitas (por exemplo, uma vulnerabilidade em algum sistema baseado em senhas pode ser a revelação da senha de um usuário autorizado a um usuário não autorizado), mas você pode ter de levantar hipóteses sobre outras possíveis vulnerabilidades.

Avaliação de riscos de proteção pode ser parte de todas as atividades do ciclo de vida, desde a engenharia de requisitos até a implantação de sistema. O processo seguido é similar ao processo de avaliação de riscos preliminar, com a adição de atividades destinadas à identificação e à avaliação de vulnerabilidades. A avaliação de vulnerabilidades identifica os ativos que podem ser afetados por elas e as relaciona a possíveis ataques ao sistema. A saída da avaliação de riscos é um conjunto de decisões de engenharia que afeta o projeto de sistema ou a sua implementação, ou limita a maneira como é usado.

Para ilustrar isso, vamos considerar que o fornecedor de tratamento de saúde tenha decidido construir seu sistema de gerenciamento de informações de pacientes de saúde mental usando um sistema de informação 'de prateleira' para manter o registro de pacientes. Esse sistema precisa ser configurado para cada tipo de clínica em que é usado. Essa decisão foi tomada porque o sistema parece oferecer funcionalidades mais extensas a um custo de desenvolvimento mais baixo e tempo de implantação mais rápido.

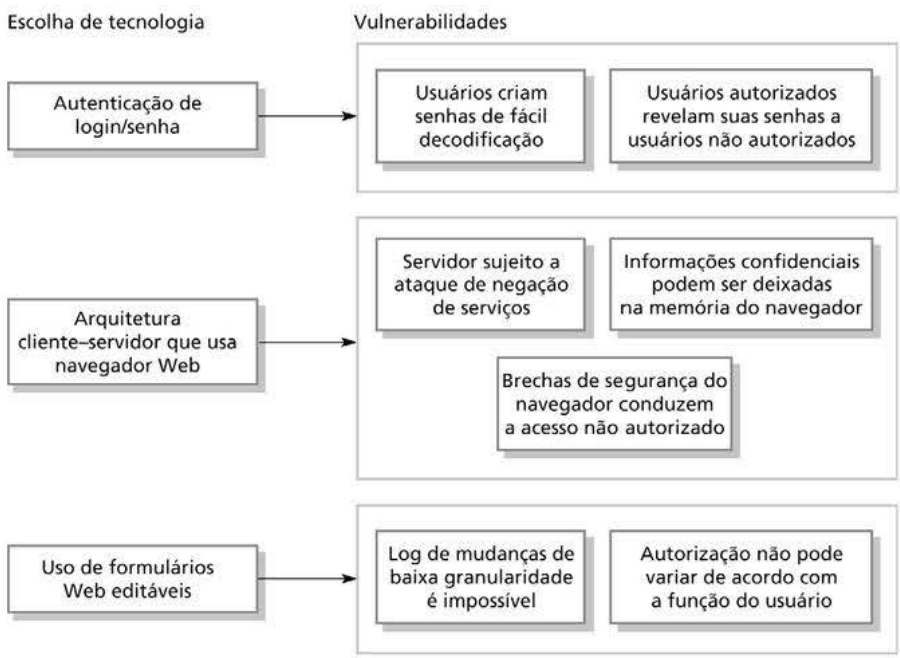
Quando você se baseia numa aplicação de um sistema existente, precisa aceitar as decisões de projeto feitas pelos desenvolvedores daquele sistema. Vamos considerar que algumas dessas decisões de projeto são:

1. Os usuários de sistema são autenticados usando uma combinação de nome de login e senha. Nenhum outro método de autenticação é tolerado.
2. A arquitetura de sistema é cliente–servidor com clientes que acessam dados por meio de um navegador Web padrão em um PC cliente.
3. A informação é apresentada aos usuários como um formulário Web editável. Eles podem mudar a informação no local e carregar a informação atualizada no servidor.

Nenhuma dessas decisões é inerentemente ruim, mas a análise de riscos de ciclo de vida revela que elas possuem vulnerabilidades associadas. Exemplos de vulnerabilidades possíveis são mostradas na Figura 30.3.

Uma vez que as vulnerabilidades foram identificadas, você precisa tomar uma decisão sobre quais passos devem ser seguidos para reduzir os riscos associados. Isso, muitas vezes, envolverá tomar decisões sobre requisitos adicionais de proteção de

Figura 30.3
Vulnerabilidades associadas a escolhas de tecnologia.



sistema ou sobre o processo operacional de uso de sistema. Não tenho espaço nesta seção para explicar todos os requisitos que poderiam ser propostos para resolver as vulnerabilidades inerentes, mas alguns exemplos de requisitos poderiam ser:

1. Um programa verificador de senhas estará disponível e funcionará diariamente. As senhas de usuários que aparecem no dicionário de sistema serão identificadas e reportadas para os administradores de sistema.
2. Acesso ao sistema será somente permitido aos computadores clientes aprovados e registrados com os administradores de sistema.
3. Todos os computadores clientes terão somente um único navegador Web instalado, conforme aprovado pelos administradores de sistema.

Como é usado um sistema de 'prateleira', não é possível incluir um verificador de senhas no sistema para assegurar que usuários não definam senhas facilmente decifráveis. No entanto, o primeiro dos requisitos significa que senhas vulneráveis podem ser identificadas de maneira razoavelmente rápida e qual ação tomar para assegurar que usuários alterem as suas senhas. O segundo e terceiro requisitos significam que os usuários irão acessar sempre o sistema por meio do mesmo navegador. Você pode decidir qual é o navegador mais seguro quando o sistema estiver implantado. Atualizações de proteção de navegador são simplificadas porque não há necessidade de alimentar diferentes sistemas.

30.3 Projeto para proteção

Conforme expliquei no Capítulo 3, proteção é uma propriedade emergente de um sistema. Ela não é, portanto, algo que possa ser adicionado ao sistema, mas precisa ser projetada no sistema antes que ele seja implementado. Ela é também, naturalmente, uma questão de implementação quando vulnerabilidades de proteção podem ser introduzidas quando o software é programado. No entanto, é possível realizar um bom projeto inseguro se a implementação for falha; é impossível fazer um mau projeto seguro no estágio de implementação.

Nesta seção, enfoco primeiro as questões de projeto de sistema, porque esse tópico não tem recebido a atenção que merece nos livros sobre proteção de computadores. Questões de proteção são, obviamente, também importantes em outros estágios do processo de software:

1. Abordei diferentes tipos de requisitos de proteção no Capítulo 9 e expliquei como a análise de riscos pode ser usada para derivar requisitos inicialmente neste capítulo, na Seção 30.2. Recomendo o artigo de Firesmith (Firesmith, 2003) como uma excelente introdução para a engenharia de requisitos de proteção.
2. Questões de implementação e erros têm impacto maior na proteção, mas são freqüentemente dependentes de uma tecnologia específica usada. Recomendo o livro do Viega e McGraw (Viega e McGraw, 2002) como uma boa introdução à programação para proteção.
3. Explico brevemente a validação de proteção no Capítulo 24 e a análise estática como técnica de verificação no Capítulo 22. Essa é uma técnica particularmente útil para descobrir vulnerabilidades de implementação (Chess e McGraw, 2004; Livshits e Lam, 2005; Evans e Larochelle, 2002; Larus et al., 2004). Para testes de proteção, recomendo o livro do Whittaker e Thompson (Whittaker e Thompson, 2004).

Nesta seção, enfoco uma série de questões gerais independentes de aplicação relevantes para assegurar o projeto de sistemas:

1. Projeto de arquitetura — como as decisões de projeto de arquitetura afetam a proteção de um sistema?
2. Boas práticas — o que é aceito como boa prática quando se projetam sistemas seguros?
3. Projeto para implantação — que apoio deve ser projetado nos sistemas para evitar a introdução de vulnerabilidades quando um sistema é implantado para uso?

Naturalmente, essas não são as únicas questões de projeto importantes para a proteção. Toda aplicação é diferente e o projeto de proteção também deve levar em conta o propósito, a importância e o ambiente operacional da aplicação. Contudo, não tenho espaço nesta seção para tratar dessas questões de projeto específicas de aplicação.

30.3.1 Projeto de arquitetura

Conforme expliquei no Capítulo 11, a escolha da arquitetura de software pode ter efeitos profundos nas propriedades emergentes de um sistema. Se uma arquitetura inadequada é usada, pode ser praticamente impossível manter a confidencialidade e a integridade de informações no sistema ou garantir o nível necessário de disponibilidade de sistema.

No projeto de uma arquitetura de sistema que mantém proteção, você precisa considerar duas questões fundamentais:

1. *Proteção* — como o sistema deve ser organizado de modo que os ativos críticos possam ser protegidos contra ataques externos?
2. *Distribuição* — como os ativos de sistema devem ser distribuídos de modo que os efeitos de um ataque bem-sucedido sejam minimizados?

Esses ativos são potencialmente conflitantes. Se você coloca todos os seus ativos num único lugar, você pode construir camadas de proteção em torno deles. No entanto, se essa proteção falha, todos os seus ativos estão comprometidos. Por outro lado, se você distribui os ativos, eles tornam-se mais onerosos de proteger e as chances de que a proteção será quebrada são maiores. No entanto, se isso acontecer, você não sofrerá um prejuízo total.

Para prover proteção em um sistema, você normalmente usa arquitetura de camadas com os ativos críticos protegidos nos níveis mais baixo do sistema e com várias camadas de proteção em torno deles. A Figura 30.4 demonstra isso para um sistema de registro de pacientes no qual os ativos críticos a serem protegidos são os registros de pacientes individuais.

A fim de acessar e modificar os registros de pacientes, um agressor precisa penetrar três camadas de sistema:

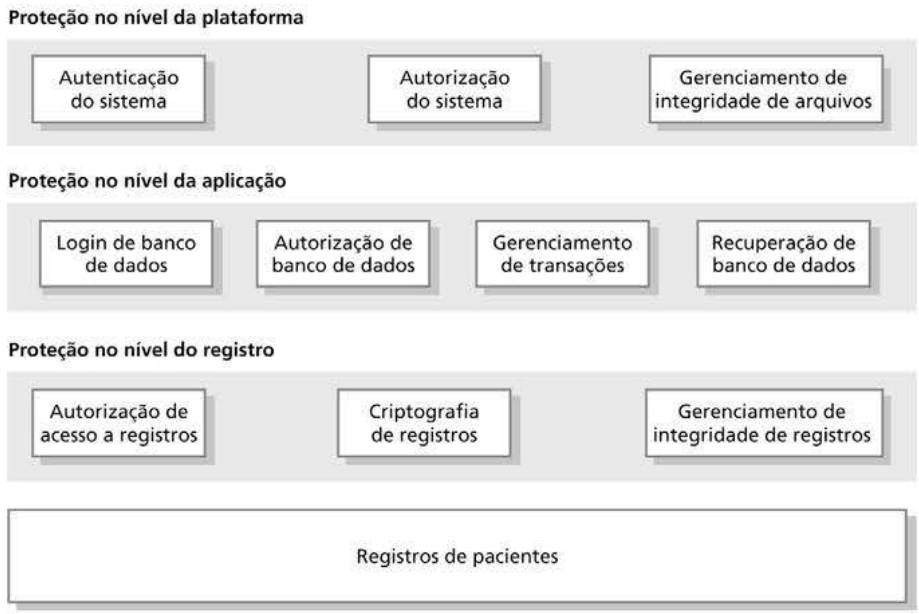
1. *Proteção no nível de plataforma*. O nível mais alto controla o acesso à plataforma na qual opera o sistema de registro de pacientes. Isso normalmente envolve a identificação de usuário em determinado computador. A plataforma incluirá também normalmente apoio para a manutenção da integridade de arquivos do sistema.
2. *Proteção no nível de aplicação*. O próximo nível de proteção é construído na própria aplicação. Ele envolve um usuário que acessa a aplicação, sendo autenticado e autorizado para realizar ações tais como visualizar ou modificar dados. O apoio de gerenciamento de integridade específico de aplicação pode estar disponível.
3. *Proteção no nível de registro*. Esse nível é invocado quando o acesso a registros específicos for necessário e envolve a verificação de se um usuário está autorizado a realizar as operações solicitadas nesses registros. A proteção nesse nível poderia também envolver criptografia para assegurar que os registros não possam ser pesquisados com o uso de um navegador de arquivos. A verificação da integridade que utiliza, por exemplo, checksums criptográficos pode detectar mudanças feitas fora dos mecanismos normais de atualização de registros.

O número de camadas de proteção de que você necessita em determinada aplicação depende da importância dos dados. Nem todas as aplicações necessitam de proteção no nível de registro e um controle de acesso não refinado é usado normalmente. Para atingir a proteção, você não deve permitir que as mesmas credenciais de usuário sejam usadas em todos os níveis. Se você tem um sistema baseado em senhas, a senha de aplicação deve ser diferente de senhas de sistema e de nível de registros.

Se a proteção dos dados for requisito crítico, uma arquitetura cliente–servidor deve ser usada com os mecanismos de proteção construídos no servidor. No entanto, se a proteção estiver comprometida, as perdas associadas com um ataque serão provavelmente altas, como serão os custos de recuperação (por exemplo, todas as credenciais de usuários deverão ser

Figura 30.4

Arquitetura de camadas de proteção.



emitidas novamente). O sistema está vulnerável a ataques de negação de serviços, que sobrecarregam o servidor e tornam impossível a alguém acessar o banco de dados do sistema.

Se você pensa que ataques de negação de serviços são os riscos principais, você pode decidir usar uma arquitetura de objeto distribuído para a aplicação. Nessa situação, ilustrada na Figura 30.5, os ativos de sistema são distribuídos por meio de um número de plataformas diferentes, com mecanismos de proteção separados e usados em cada um deles. Um ataque sobre um nó poderia significar que alguns ativos estão indisponíveis, mas seria ainda possível prover alguns serviços de sistema. Os dados podem ser replicados por meio dos nós do sistema, de modo que a recuperação aos ataques seja simplificada.

A Figura 30.5 mostra a arquitetura de um sistema bancário para negócios de produtos e fundos nos mercados de Nova York, Londres, Frankfurt e Hong Kong. O sistema está distribuído de modo que os dados sobre cada mercado sejam mantidos separadamente. Os ativos necessários para dar suporte à atividade crítica de negócios eqüitativos (contas de usuários e preços) são replicados e ficam disponíveis em todos os nós. Se um nó do sistema for atacado e tornar-se indisponível, a atividade crítica de negócios eqüitativos pode ainda receber suporte.

Um problema que pode surgir quando se projeta um sistema seguro é que o estilo de arquitetura mais adequado para prover proteção pode conflitar com outros requisitos de aplicação. Por exemplo, digamos que uma aplicação tenha um requisito absoluto para manter a confidencialidade de um grande banco de dados e outro requisito para um acesso rápido a esses dados. Satisfazer isso, na mesma arquitetura, pode ser difícil. Um alto nível de proteção sugere que as camadas de proteção são necessárias. Isso tem um custo indireto inevitável de desempenho, diminuindo, portanto, a velocidade de acesso aos dados. Se um estilo alternativo for usado, a implementação de proteção e a garantia da confidencialidade podem ser mais difíceis e onerosas. Nessas situações, você precisa discutir os conflitos inerentes com o cliente do sistema e chegar a um acordo de como isso será resolvido.

30.3.2 Diretrizes de projeto

Não há regras difíceis ou rápidas sobre como atingir a proteção de um sistema. Diferentes tipos de sistema exigem diferentes medidas técnicas para atingir o nível de proteção aceitável para o proprietário do sistema. As atitudes e os requisitos de diferentes grupos de usuários afetam profundamente o que é ou não aceitável. Por exemplo, num banco, usuários provavelmente são aceitos no nível alto de proteção e, por isso, com procedimentos de proteção mais intrometidos do que em uma universidade.

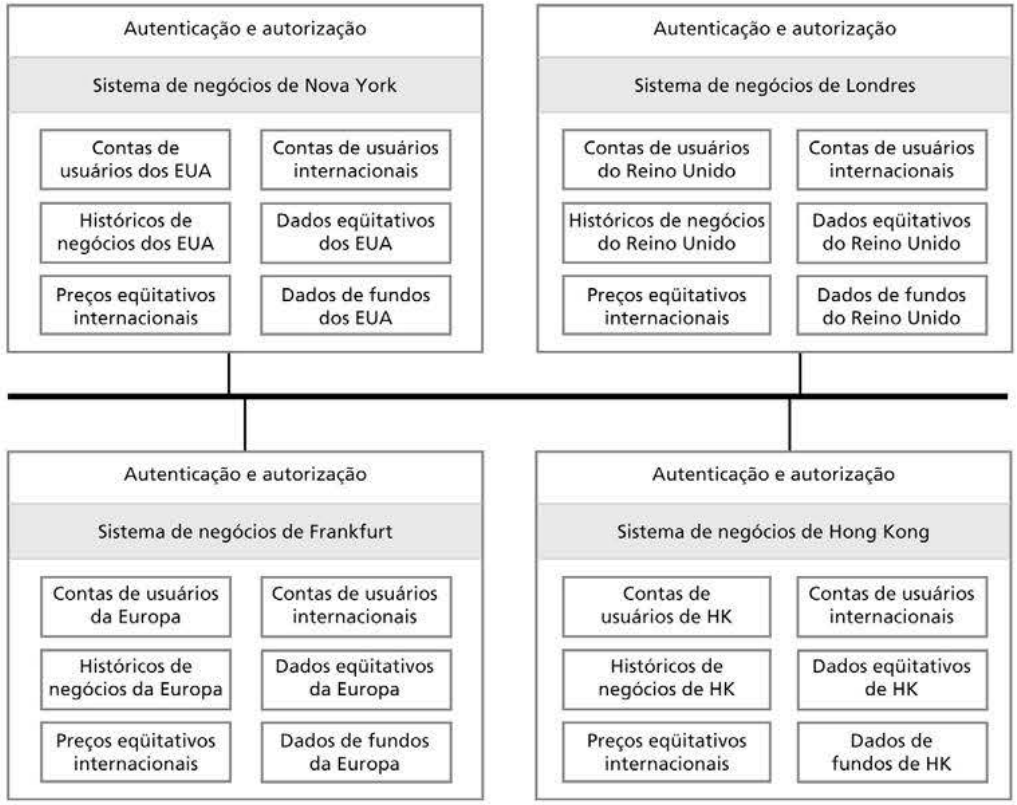


Figura 30.5
Ativos distribuídos em um sistema de negócios eqüitativo.

No entanto, existem diretrizes gerais que têm aplicabilidade ampla quando se projetam soluções de proteção de sistema que encapsulam boas práticas de projeto para sistemas seguros. Penso que diretrizes gerais de projeto para a proteção, como estes explicados nesta seção, têm dois usos principais:

1. Como um meio para aumentar a consciência sobre questões de proteção em uma equipe de engenharia de software. Os engenheiros de software muitas vezes enfocam os objetivos de curto prazo para obter o software funcionando e entregue para os clientes. É fácil eles deixarem passar questões de proteção. O conhecimento dessas diretrizes pode significar que as questões de proteção são consideradas quando as decisões de projeto de software são feitas.
2. Como uma base para um checklist de revisão que pode ser usada no processo de validação de sistema. Com base em diretrizes de nível mais alto explicadas nesta seção, questões mais específicas que explorem como a proteção foi construída em um sistema podem ser derivadas.

As dez diretrizes que explico aqui, resumidas no Quadro 30.1, foram derivadas de uma variedade de fontes diferentes (Schneier, 2000; Viega e McGraw, 2002; Wheeler, 2003). Enfoquei as diretrizes particularmente aplicáveis aos processos de especificação e de projeto de software. Princípios mais gerais, como 'Assegurar o link mais fraco no sistema', 'Mantê-lo simples' e 'Evitar proteção por meio da obscuridade' são também importantes, mas são diretamente menos relevantes para tomada de decisão de engenharia.

Diretriz 1: Basear as decisões de proteção em uma política de proteção explícita

Uma política de proteção é uma declaração de alto nível que estabelece condições de proteção fundamentais para uma organização. Ela define o 'que' da proteção, no lugar de 'como'. A política não deve definir mecanismos usados para prover e fazer cumprir a proteção. Em princípio, todos os aspectos da política de proteção devem ser refletidos nos requisitos de sistema. Na prática, especialmente se um processo de desenvolvimento rápido de aplicações for usado, isso é improvável de acontecer. Os projetistas devem, portanto, consultar a política de proteção quando ela provê um framework para tomar e avaliar decisões de projeto.

Por exemplo, digamos que você esteja projetando um sistema de controle de acesso para um sistema de informações de pacientes de tratamento de doenças mentais introduzido anteriormente no capítulo. A política de proteção do hospital pode estabelecer que somente o pessoal clínico credenciado pode modificar registros eletrônicos de pacientes. O seu sistema precisa, portanto, incluir mecanismos que verifiquem o credenciamento de qualquer pessoa que tente modificar o sistema e que rejeite modificações de pessoas não credenciadas.

Diretriz 2: Evitar um ponto único de falha

Em qualquer sistema crítico, é uma boa prática de projeto tentar evitar um ponto único de falha. Isso significa que uma falha única em uma parte do sistema não deve resultar na falha geral de sistemas. Em termos de proteção, isso significa que você não deve contar com um único mecanismo para assegurar a proteção, mas deve empregar várias técnicas distintas. Isso é algumas vezes chamado 'defesa em profundidade'.

Por exemplo, se você usa uma senha para autenticar usuários para um sistema, você poderia também incluir um mecanismo de autenticação do tipo desafio/resposta no qual os usuários precisassem registrar previamente as perguntas e as respostas no sistema. Depois da autenticação da senha, eles deveriam então responder às perguntas corretamente antes de ser permitido o acesso. Para proteger a integridade de dados no sistema, você poderia manter um log de todas as mudanças feitas nos dados, de modo que, se ocorrer uma falha, você pode examinar o log para recriar o conjunto de dados. Você poderia também manter uma cópia de todos os dados modificados antes que a mudança fosse feita.

Quadro 30.1

Diretrizes de projeto para engenharia de sistemas protegidos.

	Diretriz
1	Basear as decisões de proteção em uma política de proteção explícita
2	Evitar um ponto único de falha
3	Falhar de maneira protegida
4	Equilibrar proteção e facilidade de uso
5	Estar ciente da possibilidade de engenharia social
6	Usar redundância e diversidade para reduzir riscos
7	Validar todas as entradas
8	Compartimentar seus ativos
9	Projetar para implantação
10	Projetar para capacidade de recuperação

Diretriz 3: Falhar de maneira protegida

As falhas de sistema de algum tipo são inevitáveis em todos os sistemas e, do mesmo modo que sistemas críticos de segurança devem sempre falhar de maneira segura (*fail-safe*), sistemas críticos de proteção devem sempre falhar de maneira protegida (*fail-secure*). Você não deve ter procedimentos de contingência usados quando o sistema falha que sejam menos seguros que o sistema em si. Nenhuma falha de sistema deve possibilitar a um agressor acesso aos dados que não seria normalmente permitido.

Por exemplo, no sistema de informações de pacientes, sugeri um requisito em que os dados de pacientes deveriam ser baixados para um sistema cliente no início de uma sessão clínica. Isso acelera o acesso e significa que o acesso será possível se o servidor estiver indisponível. Normalmente, o servidor exclui esses dados no fim da sessão clínica. No entanto, se o servidor falhou, há a possibilidade de que as informações serão mantidas no cliente. Uma abordagem de falha em proteção (*fail-secure*) nessas circunstâncias pode ser criptografar os dados de pacientes no cliente. Se esses dados não forem excluídos, não poderão ser lidos por pessoas não autorizadas.

Diretriz 4: Equilibrar proteção e facilidade de uso

As demandas de proteção e facilidade de uso são muitas vezes contraditórias. Para manter o sistema protegido, você deve introduzir uma série de verificações para assegurar que usuários estejam autorizados a usar o sistema e para garantir que eles estão agindo de acordo com as políticas de proteção. Tudo isso gera inevitavelmente demandas aos usuários — eles podem ter de lembrar-se de nomes de login e senhas, usar somente o sistema com base em determinados computadores etc. Essas demandas tomam mais tempo dos usuários para iniciar e usar o sistema eficientemente. À medida que você adiciona características de proteção a um sistema, é inevitável que ele se torne menos fácil de usar.

Portanto, chega-se a um ponto em que é contraproducente manter a adição de novas características de proteção à custa da facilidade de uso. Por exemplo, se você exigir que os usuários insiram várias senhas ou alterem suas senhas para impossibilitar que strings de caracteres sejam memorizadas em intervalos freqüentes, eles simplesmente escreverão suas senhas. Um agressor (especialmente um intruso) poderá, então, encontrar as senhas e obter acesso ao sistema.

Diretriz 5: Estar ciente da possibilidade de engenharia social

Engenharia social significa encontrar maneiras de ludibriar usuários de sistema credenciados a fim de revelar informações secretas. Essas abordagens aproveitam-se da boa vontade de pessoas em ajudar e da confiança delas na autoridade em uma organização. Por exemplo, um empregado júnior pode ser contatado por alguém fingindo ser um gerente sênior que reclama estar incapacitado em acessar um sistema. Se ele pergunta sobre detalhes de proteção, o funcionário júnior pode ficar relutante em recusar essa solicitação. Contudo, ao revelar detalhes de proteção, pode facilitar o acesso de alguma pessoa não autorizada ao sistema.

Do ponto de vista de projeto, enfrentar a engenharia social é muito difícil. Se a proteção for absolutamente crítica, você não deve contar com mecanismos de autenticação de usuários que estão relacionados com os nomes de login e senhas; deve usar técnicas de autenticação mais robustas tais como certificados digitais. Mecanismos de registro que acompanham a localização e a identificação dos usuários e os programas de análise de registros podem também ser úteis, na medida em que permitem detectar brechas de proteção.

Diretriz 6: Usar redundância e diversidade para reduzir riscos

Redundância significa que você mantém mais de uma versão de software ou de dados no sistema. Diversidade, quando aplicada ao software, significa que versões diferentes não devem ser baseadas na mesma plataforma ou usar as mesmas tecnologias. Portanto, uma vulnerabilidade de plataforma ou de tecnologia não afetará todas as versões e, dessa maneira, conduzirá a falhas de modo comum.

Já expliquei exemplos de redundância — manutenção de informações de pacientes em ambos, no servidor e no cliente, no sistema de tratamento de saúde mental e no sistema de negócios eqüitativos distribuído na Figura 30.5. No sistema de registros de pacientes você pode usar sistemas operacionais diferentes no cliente e no servidor (veja Linux no servidor; Windows no cliente) de modo a assegurar que um ataque baseado na vulnerabilidade do sistema operacional não afete o servidor e o cliente simultaneamente.

Diretriz 7: Validar todas as entradas

Um ataque comum ao sistema envolve o fornecimento de entradas inesperadas ao sistema que causam um comportamento imprevisto. Isso pode simplesmente provocar a quebra de um sistema, resultando na perda de serviço, ou as entradas poderiam ser feitas com códigos maliciosos interpretados pelo sistema. Vulnerabilidades de overflow de buffer, surgidas primeiro como um worm de Internet (Spafford, 1989) e freqüentemente usadas por agressores (Berghel, 2001), podem ser disparadas usando um string longo de entrada. O chamado 'veneno SQL' em que um usuário malicioso entra em um fragmento SQL, interpretado por um servidor, é outro ataque muito comum.

Você pode evitar muito desses problemas se projetar uma validação de entradas no seu sistema. Essencialmente, você nunca deve aceitar nenhuma entrada sem aplicar algumas verificações sobre ela. Como parte dos requisitos, devem-se definir as verificações que seriam aplicáveis. Você deve usar o conhecimento da entrada para definir essas verificações. Por exemplo, se um sobrenome deve ser inserido, você pode verificar se não foram incluídos espaços e se somente um hífen é usado como pontuação. Você pode também verificar o número de caracteres de entrada e rejeitar entradas evidentemente muito longas. Por exemplo, ninguém possui um sobrenome com mais de 70 caracteres e nenhum endereço é maior do que 100 caracteres. Conforme expliquei no Capítulo 16, se usar menus para apresentar as entradas permitidas, você evita alguns dos problemas de validação de entradas.

Diretriz 8: Compartimentar seus ativos

Compartimentalização significa que você não deve prover todo ou nenhum acesso à informação no sistema. Em vez disso, deve organizar a informação no sistema de modo que os usuários somente tenham acesso à informação de que necessitam, no lugar de toda a informação do sistema. Isso significa que o efeito de um ataque pode ser contido. Algumas informações podem ser perdidas ou danificadas, mas é improvável que toda a informação do sistema seja afetada.

Por exemplo, em um sistema de informação de pacientes, você deve projetar o sistema de modo que, em qualquer clínica, o pessoal clínico tenha normalmente acesso somente aos registros de pacientes que tenham uma consulta marcada naquela clínica. Eles não devem ter normalmente acesso a todos os pacientes do sistema. Você também pode precisar de mecanismos no sistema para conceder acessos inesperados — digamos a um paciente gravemente doente que requeira tratamento urgente sem consulta marcada. Nessas circunstâncias, você pode usar um mecanismo de proteção alternativo para ignorar a compartimentalização do sistema.

Diretriz 9: Projetar para implantação

Muitos dos problemas de proteção surgem porque o sistema não está configurado corretamente quando é implantado no seu ambiente operacional. Você deve, portanto, projetar sempre seu sistema de modo que os recursos sejam incluídos para simplificar a implantação e para verificar erros potenciais de configuração e omissões do sistema implantado. Esse é um tópico importante que abordo na Seção 30.3.3.

Diretriz 10: Projetar para capacidade de recuperação

Independentemente de quanto esforço você despende na manutenção de proteção de sistema, você deve sempre projetar o seu sistema com a hipótese de que uma falha de proteção possa ocorrer. Portanto, você deve pensar em como se recuperar de possíveis falhas e restaurar o sistema para um estado operacional seguro.

Por exemplo, digamos que uma pessoa não autorizada de fora do hospital tenha conseguido acesso ao sistema de registros de pacientes e você não descobriu como ele tinha obtido uma combinação válida de login/senha. Você necessita, portanto, alterar os detalhes de senha de todos os usuários credenciados e estar certo de que pessoas não autorizadas não tenham acesso ao mecanismo de alteração de senhas. Você precisa, portanto, projetar seu sistema para negar acesso a qualquer pessoa até que todos os usuários tenham alterado suas senhas e autenticar usuários reais, apesar de as senhas poderem não ser seguras. Uma maneira de fazer isso é usar um mecanismo do tipo desafio/resposta, no qual os usuários precisam responder a questões para as quais possuam respostas pré-registradas. Isso seria invocado somente quando as senhas fossem alteradas.

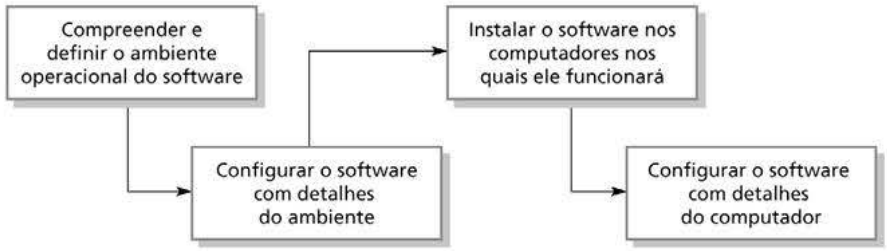
30.3.3 Projetar para implantação

A implantação de um sistema envolve a configuração do software para operar em um ambiente operacional, instalando o sistema nos computadores desse ambiente para, então, configurar o sistema instalado nesses computadores (Figura 30.6). A configuração pode ser tão simples quanto estabelecer alguns parâmetros incorporados ao software para refletir as preferências de usuários ou tão complexa quanto definir modelos e regras de negócios que governam a execução do software.

É nesse estágio do processo de software que as vulnerabilidades de software são freqüentemente introduzidas acidentalmente. Por exemplo, o software tem de ser muitas vezes configurado com uma lista de usuários autorizados e, quando ele for entregue, essa lista simplesmente consistirá de um login genérico de administrador, como 'admin', e de uma senha default, como 'senha'. Isso torna fácil para um administrador estabelecer o sistema. A sua primeira ação deve ser introduzir um novo nome de login e senha, e excluir o nome genérico de nome de login. Entretanto, é fácil esquecer-se de fazer isso. Um agressor que conheça o login default pode ser capaz de obter acesso privilegiado ao sistema.

A configuração e a implantação são freqüentemente vistas como questões de administração de sistema e, portanto, fora do escopo de processos de engenharia de software. Certamente, boas práticas de gerenciamento podem evitar muitos problemas de proteção que surgem de erros de configuração e de implantação. No entanto, os projetistas de software têm a responsabilidade de 'projetar para implantação'. Você deve sempre prover apoio de recursos incorporados para implantação que reduza a probabilidade de que os administradores de sistema (ou usuários) cometerão erros ao configurarem o software.

Figura 30.6
Implantação de software.



Recomendo quatro maneiras pelas quais você pode incorporar apoio à implantação em um sistema:

1. *Incluir apoio para visualização e análise de configurações.* Você deve sempre incluir recursos no sistema que permitam aos administradores ou aos usuários autorizados examinar a configuração atual do sistema. Esse recurso é, surpreendentemente, omitido em muitos sistemas de software, e os usuários ficam frustrados pelas dificuldades de encontrar os parâmetros de configuração. Por exemplo, se você usa versões do Microsoft Word disponível em 2005, saberá que é impossível ver ou imprimir os parâmetros de todas as suas preferências numa única tela. Se um administrador pode ter uma imagem completa de uma configuração, é provável que possa destacar erros ou omissões. De preferência, uma tela de configuração deve também destacar aspectos de configuração potencialmente inseguros — por exemplo, se uma senha não foi estabelecida.
2. *Minimizar privilégios default.* Você deve projetar o software de modo que a configuração default do sistema forneça privilégios essenciais mínimos. Portanto, o dano que um agressor pode causar pode ser limitado. Por exemplo, a autenticação default de administrador de sistema deve somente permitir o acesso para um programa que permita a um administrador estabelecer novas credenciais. Ele não deve permitir o acesso a quaisquer outros recursos de sistema. Uma vez que as novas credenciais tenham sido estabelecidas, o login e a senha default devem ser excluídos automaticamente.
3. *Localizar parâmetros de configuração.* Ao projetar o apoio de configuração de sistema, você deve assegurar que tudo na configuração que pertence à mesma parte de um sistema seja estabelecido no mesmo lugar. Para usar o exemplo do Microsoft Word novamente, na versão do Word que uso, posso estabelecer algumas informações de proteção, tais como uma senha para controlar acesso ao documento, usando o menu Preferências/Proteção, mas outra informação é estabelecida no menu Ferramentas/Proteger Documento. Se a informação de configuração não é localizada, é fácil esquecer-se de estabelecer ou, em alguns casos, sequer ficar ciente de que alguns recursos de proteção estão incluídos no sistema.
4. *Prover maneiras simples para reparar vulnerabilidades de proteção.* Você deve incluir mecanismos simples para atualizar o sistema reparando as vulnerabilidades de proteção descobertas. Esses mecanismos podem incluir verificação automática de atualizações de proteção e carregamento automático dessas atualizações tão logo estejam disponíveis. Você deve também considerar como atualizar centenas de PCs nos quais o seu sistema esteja instalado.

30.4 Capacidade de sobrevivência de sistemas

Até aqui, expliquei a engenharia de proteção da perspectiva de uma aplicação que está em desenvolvimento. O adquirente e o desenvolvedor do sistema têm controle sobre todos os aspectos do sistema que poderiam ser atacados. Na realidade, conforme sugeri na Figura 30.1, sistemas modernos distribuídos inevitavelmente contam com COTS e componentes reusáveis desenvolvidos separadamente. Suas características de proteção podem ser serviços Web externos e infra-estruturas de rede que estão fora do controle da aplicação.

Isso significa que, independentemente de quanta atenção for dedicada à proteção, não se pode garantir que o sistema será capaz de resistir a ataques externos. Conseqüentemente, para sistemas complexos em rede, deve-se considerar que a invasão é possível e que a integridade do sistema não pode ser garantida. Você deve, portanto, pensar em como fazer o sistema resistente de modo que ele sobreviva para fornecer serviços essenciais aos usuários.

A capacidade de sobrevivência (Westmark, 2004) é uma propriedade emergente de um sistema como um todo, e não uma propriedade de componentes individuais, que não podem sobreviver sozinhos. A capacidade de sobrevivência de um sistema reflete sua habilidade em continuar a fornecer os serviços de missão crítica ou de negócios essenciais para usuários legítimos enquanto ele estiver sob ataque ou depois que parte do sistema foi danificada em conseqüência de um ataque ou falha de sistema.

O trabalho em capacidade de sobrevivência de sistema foi estimulado pelo fato de que nossa vida social e econômica é dependente de uma infra-estrutura crítica controlada por computador. Isso inclui a infra-estrutura para a entrega de serviços (energia, água, gás etc.) e, igualmente crítica, a infra-estrutura para entrega e gerenciamento de informações (telefones, Internet, serviços postais etc.). No entanto, a capacidade de sobrevivência não é simplesmente uma questão crítica de infra-estrutura. Qualquer organização que conte com sistemas computacionais críticos em rede deve estar preocupada em como o seu negócio será afetado se seus sistemas não sobreviverem a um ataque malicioso ou a uma falha catastrófica de sistema. Portanto, para sistemas críticos de negócios, o projeto e a análise de capacidade de sobrevivência devem ser parte do processo de engenharia de proteção.

A manutenção da disponibilidade de serviços críticos é essencial para a capacidade de sobrevivência. Isso significa que você deve saber:

- quais serviços de sistema são mais críticos para um negócio;
- como esses serviços poderiam ficar comprometidos;
- qual é a mínima qualidade de serviço que deve ser mantida;
- como esses serviços podem ser protegidos;
- como você pode se recuperar rapidamente se os serviços se tornarem indisponíveis.

Por exemplo, em um sistema que trata de despacho de ambulâncias em resposta a chamadas de emergência, os serviços críticos são os dedicados ao atendimento de chamadas e ao despacho de ambulâncias para as emergências médicas. Outros serviços, como o registro de chamadas e o gerenciamento de localização de ambulâncias, são menos críticos porque não exigem processamento de tempo real ou porque mecanismos alternativos podem ser usados. Por exemplo, para descobrir a localização de uma ambulância, você pode ligar para o pessoal da ambulância e perguntar onde eles estão.

Ellison et al. (1999; 1999a; 2002) desenvolveram um método de análise chamado Análise de Sobrevivência de Sistemas (*Survivable Systems Analysis*). Ele é usado para avaliar as vulnerabilidades no sistema para apoiar o projeto de arquiteturas de sistemas e características que promovam a capacidade de sobrevivência do sistema. Eles argumentam que atingir a capacidade de sobrevivência depende de três estratégias complementares:

1. *Resistência:* evitar problemas pela implementação de capacidades no sistema para repelir ataques. Por exemplo, um sistema pode usar certificados digitais para autenticar usuários, dificultando assim que usuários não autorizados obtenham acesso.
2. *Reconhecimento:* detectar problemas pela implementação de capacidades no sistema para detectar ataques e falhas e avaliar os danos resultantes. Por exemplo, os checksums podem ser associados a dados críticos, de modo que o corrompimento desses dados possa ser detectado.
3. *Recuperação:* tolerar problemas pela implementação de capacidades no sistema para fornecer serviços essenciais durante o ataque e recuperar a plena funcionalidade após o ataque. Por exemplo, mecanismos de tolerância a defeitos, usando diversas implementações de uma mesma funcionalidade, podem ser incluídos para lidar com uma perda de serviço de uma parte do sistema.

A análise de sobrevivência de sistemas é um processo de quatro estágios (Figura 30.7) que avalia os requisitos e a arquitetura de sistema atual ou proposto, identifica serviços críticos, cenários de ataque e pontos fracos de sistema e propõe mudanças para melhorar a capacidade de sobrevivência de um sistema. As atividades principais em cada um desses estágios são:

Figura 30.7

Estágios de análise de capacidade de sobrevivência.

1. *Compreensão de sistema*. Para um sistema atual ou proposto, revisar os objetivos (algumas vezes chamados de objetivos de missão), os requisitos e a arquitetura do sistema.
2. *Identificação de serviços críticos*. Os serviços que devem ser mantidos e os componentes necessários para mantê-los são identificados.
3. *Simulação de ataques*. Cenários ou casos de uso de possíveis ataques são identificados junto com componentes de sistema que seriam afetados por esses ataques.
4. *Análise de capacidade de sobrevivência*. Componentes essenciais e que possam ficar comprometidos por um ataque são identificados e estratégias de capacidade de sobrevivência baseadas em resistência, reconhecimento e recuperação são identificadas.

Ellison et al. (1999) apresentam um excelente estudo de caso do método baseado em um sistema para apoio ao tratamento de saúde mental. Uso o sistema de negócios eqüitativos, mostrado na Figura 30.5, para ilustrar algumas das características da análise de capacidade de sobrevivência.

Como você pode ver na Figura 30.5, esse sistema já tem alguma provisão para a capacidade de sobrevivência. Contas de usuário e preços eqüitativos são replicados entre os servidores, de modo que os pedidos possam ser inseridos se o servidor local estiver indisponível. Vamos considerar que a capacidade de usuários autorizados de colocar pedidos de produtos é o serviço principal que deve ser mantido. Para assegurar que os usuários confiem no sistema, é essencial que a integridade seja mantida. Os pedidos devem ser precisos e refletir as vendas reais ou as compras feitas por um usuário de sistema.

Para manter esse serviço de pedidos, são usados três componentes do sistema:

1. *Autenticação de usuários*. Permite que os usuários autorizados façam o logon no sistema.
2. *Cotação de preços*. Permite que o preço de compra e venda de produto seja cotado.
3. *Envio de pedidos*. Permite que pedidos de compra e venda a determinado preço sejam feitos.

Esses componentes fazem, obviamente, uso de ativos de dados essenciais, tais como contas de usuários e um banco de dados de transações de pedidos.

Existem muitos tipos diferentes de ataques que poderiam ser feitos no sistema. Vamos considerar duas possibilidades aqui:

1. Um usuário malicioso tem raiva de um usuário credenciado no sistema. Ele obtém acesso ao sistema usando as suas credenciais. Os pedidos maliciosos são enviados, produtos são comprados e vendidos, de maneira que são causados problemas para o usuário autorizado.
2. Um usuário que não é autorizado corrompe o banco de dados de transações por ganhar a permissão de gerar comandos SQL diretamente. A reconciliação de vendas e compras se torna, portanto, impossível.

A Tabela 30.5 mostra exemplos de estratégias de resistência, reconhecimento e recuperação que poderiam ser usadas para conter esses ataques.

Tabela 30.5 Análise de capacidade de sobrevivência em um sistema de negócios eqüitativos

Ataque	Resistência	Reconhecimento	Recuperação
Usuário não autorizado envia pedidos maliciosos	Exigir uma senha de negócio diferente da senha de login para enviar pedidos	Enviar cópia do pedido por e-mail para um usuário autorizado com número de telefone de contato (assim eles podem detectar pedidos maliciosos). Manter o histórico de pedidos do usuário e verificar padrões de transação incomuns.	Fornecer mecanismo para 'desfazer' automaticamente as transações e restaurar contas de usuários. Ressarcir usuários por perdas devido às transações maliciosas. Proteger-se contra as conseqüências de perdas.
Corrompimento de banco de dados de transações	Exigir que os usuários privilegiados sejam autorizados a usar um mecanismo de autenticação robusto, como certificados digitais.	Manter cópias somente para leitura de transações de um escritório em um servidor internacional. Comparar periodicamente as transações para verificar se foram corrompidas. Manter checksum criptográfico em todos os registros de transações para detectar corrompimento.	Recuperar o banco de dados com base em cópias de back-up. Fornecer um mecanismo para reproduzir as transações de um período específico para recriar banco de dados de transações.

Acrescentar capacidade de sobrevivência, naturalmente, custa dinheiro. As empresas podem ficar relutantes em investir em capacidade de sobrevivência se elas nunca sofreram um ataque sério ou tiveram perdas associadas. No entanto, é ainda melhor comprar boas fechaduras e um alarme antes de a casa ser arrombada; é melhor investir em capacidade de sobrevivência antes de um ataque bem-sucedido. A análise de capacidade de sobrevivência não é ainda parte integrante dos processos de engenharia de software, mas, à medida que mais e mais sistemas tornam-se críticos do ponto de vista de mercado, tais análises provavelmente vão se tornar mais amplamente usadas.

PONTOS-CHAVE

- A engenharia de proteção enfoca como desenvolver e manter sistemas de software que possam resistir a ataques maliciosos com a intenção de danificar um sistema baseado em computadores ou seus dados.
- As ameaças de proteção podem ser ameaças à confiabilidade, integridade ou disponibilidade de um sistema e a seus dados.
- O gerenciamento de riscos de proteção envolve a avaliação de perdas que podem resultar de um ataque em um sistema e a derivação de requisitos de proteção que têm como objetivo a eliminação ou a redução dessas perdas.
- O projeto para proteção envolve o projeto de uma arquitetura de sistema protegida, seguindo boas práticas de projeto de sistemas protegidos e incluindo funcionalidades para minimizar a possibilidade de introdução de vulnerabilidades de proteção quando o sistema for implantado.
- As questões principais ao projetar uma arquitetura de sistemas protegida incluem a organização de uma estrutura de sistema para proteger os ativos principais e a distribuição dos ativos de sistema para minimizar as perdas de um ataque bem-sucedido.
- As diretrizes gerais de projetos de proteção sensibilizam os projetistas de sistemas quanto a questões de proteção que eles podem não ter considerado. As diretrizes fornecem uma base para criar checklists para revisão de proteção.
- Para apoiar a implantação de proteção, você deve prover um meio de apresentar e analisar as configurações de sistema, localizar os parâmetros de configurações de modo que configurações importantes não sejam esquecidas, minimizar privilégios default definidos para usuários de sistema e fornecer meios para reparar as vulnerabilidades de proteção.
- A capacidade de sobrevivência de sistemas reflete a habilidade do sistema em continuar a fornecer os serviços de negócio essenciais ou de missão crítica para usuários legítimos quando sob ataque ou após uma parte do sistema ter sido danificada.

LEITURAS SUGERIDAS

Security engineering: a guide to building dependable distributed systems. É uma explicação minuciosa e abrangente dos problemas de construção de sistemas protegidos. O foco recai nos sistemas em vez de na engenharia de software com ampla abrangência de hardware e rede. Exemplos excelentes retirados de falhas reais de sistema (R. Anderson, John Wiley & Sons, 2001).

Building secure software: how to avoid security problems the right way. Um bom livro prático que abrange proteção sob a perspectiva de programação (J. Viega e G. McGraw, Addison-Wesley, 2002).

"Survivable network system analysis: a case study". Excelente artigo que apresenta a noção de capacidade de sobrevivência de sistema e usa um estudo de caso de um sistema de registro de tratamento de saúde mental para ilustrar a aplicação de um método de capacidade de sobrevivência (R. J. Ellison, R. C. Linger, T. Longstaff e N. R. Mead, *IEEE Software*, 16 (4), julho/agosto de 1999).

EXERCÍCIOS

30.1 Explique as diferenças importantes entre a engenharia de proteção de aplicações e a engenharia de proteção de infra-estrutura.

30.2 O LIBSYS é um sistema para a distribuição de documentos explicado nos capítulos anteriores. Para o sistema LIBSYS, sugira um exemplo de ativo, exposição, vulnerabilidade, ataque, ameaça e controle.

30.3 Explique por que existe a necessidade de avaliação de riscos preliminar e de avaliação de riscos de ciclo de vida durante o desenvolvimento de um sistema.

30.4 Usando suas respostas para a questão 2, sobre o sistema LIBSYS, avalie os riscos associados a esse sistema e proponha dois requisitos que possam reduzi-los.

30.5 Explique, usando uma analogia extraída de fora da engenharia de software, por que deve ser usada uma abordagem em camadas para proteção de ativos.

30.6 Explique por que é importante usar tecnologias diversas para dar suporte a sistemas distribuídos em situações em que a disponibilidade do sistema é crítica.

30.7 O que é engenharia social? Por que é difícil se proteger contra ela em grandes organizações?

30.8 Para qualquer sistema de software comercial usado (por exemplo, Microsoft Word), analise os recursos de configuração incluídos e explique alguns problemas que você pode encontrar.

30.9 Explique como as estratégias complementares de resistência, reconhecimento e recuperação podem ser usadas para aumentar a capacidade de sobrevivência de um sistema.

30.10 Para o sistema de negócios eqüitativos explicado na Seção 30.4 e ilustrado na Figura 30.5, sugira dois novos ataques plausíveis ao sistema e proponha estratégias possíveis para contê-los.

31

Engenharia de software orientada a serviços

Objetivos

O objetivo deste capítulo é apresentar a engenharia de software orientada a serviços, importante abordagem em crescimento para o desenvolvimento das aplicações de negócios. Após ler este capítulo, você:

- poderá compreender as noções básicas e os padrões de Web service, e como eles podem apoiar a computação interorganizacional;
- compreenderá o processo da engenharia de serviços que tem a intenção de produzir Web services reusáveis;
- terá sido apresentado às noções da composição de serviços como um meio de desenvolvimento de aplicações orientadas a serviços;
- compreenderá como os modelos de processo de negócios podem ser usados como referência para os projetos de sistemas orientados a serviço.

Conteúdo

31.1 Serviços como componentes reusáveis
31.2 Engenharia de serviços
31.3 Desenvolvimento de software como serviços

No Capítulo 12 apresentei a noção de arquiteturas orientadas a serviços como um meio para facilitar a computação interorganizacional. Essencialmente, as arquiteturas orientadas a serviços (SOA — *Service-Oriented Architectures*) são um caminho para o desenvolvimento de sistemas distribuídos nos quais os componentes desses sistemas são serviços dedicados. Os serviços podem ser executados em computadores distribuídos geograficamente. Protocolos padronizados foram projetados para apoiar troca de serviços de comunicação e de informações. Conseqüentemente, os serviços são plataformas e linguagens de implementação independentes. Sistemas de software podem ser construídos usando serviços de provedores diferentes sem nenhuma ligação de interação entre esses serviços.

A Figura 31.1 ilustra como os Web services são usados. Os provedores de serviços projetam e implementam serviços e os especificam em uma linguagem chamada WSDL (explicada mais adiante). Eles também publicam informações sobre esses serviços em um registro de acesso geral usando um padrão de publicação chamado UDDI. Os solicitantes de serviços (algumas vezes chamados de clientes de serviços), que desejam fazer uso de um serviço, buscam o registro UDDI para

Figura 31.1
Arquitetura orientada a serviços.

descobrir a especificação desse serviço e para localizar um provedor de serviços. Eles podem então unir as suas aplicações para um serviço específico e se comunicar com ele, usando geralmente um protocolo chamado de SOAP.

A arquitetura orientada a serviços é atualmente reconhecida como desenvolvimento significativo, especialmente para sistemas de aplicações de negócios. Ela permite flexibilidade como serviços que possam ser fornecidos localmente ou terceirizados para provedores externos. Os serviços podem ser implementados em qualquer linguagem de programação. Por meio do empacotamento de sistemas legados (veja o Capítulo 21) como serviços, as empresas podem preservar seus investimentos com softwares valiosos e mantê-los disponíveis para uma grande gama de aplicações. O SOA permite que plataformas e tecnologias de implementação diferentes possam ser usadas em diferentes partes das empresas para se interoperarem. Mais fundamentalmente, talvez, a construção de aplicações baseadas em serviços permite que empresas e demais organizações cooperem e façam uso de outras funções de negócio. Portanto, os sistemas que envolvem extensa troca de informações por meio das fronteiras da empresa, como sistemas da cadeia de suprimentos, nos quais uma empresa encomenda produtos de outras, podem facilmente ser automatizados.

Talvez a razão principal para o sucesso das arquiteturas orientadas a serviços é o fato de que, a princípio, tem ocorrido um processo ativo de padronização trabalhando lado a lado com os desenvolvimentos técnicos. Todas as principais empresas de hardware e de software estão comprometidas com esses padrões. Como resultado, as arquiteturas orientadas a serviços não sofreram com incompatibilidades, que normalmente acontecem com inovações tecnológicas, nas quais diferentes fornecedores mantêm as suas versões proprietárias de tecnologia. Por essa razão, problemas, como modelos múltiplos de incompatibilidades de componentes em CBSE, que expliquei no Capítulo 19, não surgiram em desenvolvimentos de sistemas orientados a serviços.

A Figura 31.2 mostra o conjunto de padrões principais definidos para apoiar os Web services. Em princípio, uma abordagem orientada a serviços pode ser aplicada em situações em que outros protocolos são usados; na prática, os Web services são dominantes. Os Web services não dependem de nenhum protocolo de transporte específico para a troca de informações, embora, na prática, os protocolos HTTP e HTTPS sejam comumente usados.

Os protocolos de Web services cobrem todos os aspectos das arquiteturas orientadas a serviços desde os mecanismos básicos para os serviços de troca de informações (SOAP) até padrões de linguagens de programação (WS-BPEL). Esses padrões são todos baseados em XML, notação que pode ser lida por homens e máquinas que permite uma definição de estrutura de dados em que o texto é marcado com um identificador significativo. O XML possui uma gama de tecnologias de apoio, como o XSD para definição de esquemas, usado para estender e manipular as descrições XML. Erl (Erl, 2004) fornece um bom resumo das tecnologias XML e seu papel nos Web services.

Resumidamente, o padrões principais das arquiteturas orientadas a serviços são:

1. *SOAP.* Este é um padrão de troca de mensagens que dá suporte à comunicação entre serviços. Ele define os componentes essenciais e opcionais das mensagens passadas entre serviços.

Figura 31.2
Padrões de serviços Web.

2. *WSDL.* O padrão linguagem de definição de serviço Web (WSDL — *Web Service Definition Language*) estabelece o meio pelo qual os provedores de serviços devem definir a interface para esses serviços. Essencialmente, ele permite que a interface de um serviço (operações de serviços, parâmetros e seus tipos) e suas ligações sejam definidas de maneira padronizada.
3. *UDDI.* O padrão descrição, descoberta e integração universais (UDDI — *Universal Description, Discovery and Integration*) define os componentes de uma especificação de serviços que pode ser usada para descobrir a existência de um serviço. Esses componentes incluem informações sobre o provedor de serviços, os serviços fornecidos, a localização da descrição de serviços (usualmente expressa em WSDL) e informações sobre relacionamentos de negócio. O UDDI registra os usuários potenciais de um serviço capazes de descobrir quais serviços estão disponíveis.
4. *WS-BPEL.* É um padrão para uma linguagem de workflow usado para definir programas de processo que envolvem vários serviços diferentes. Explico a noção de programas de processo na Seção 31.3.

Esses padrões principais recebem suporte de uma gama de padrões de apoio que foca os aspectos mais especializados do SOA. Existe grande quantidade de padrões de apoio porque eles têm a intenção de apoiar o SOA em diferentes tipos de aplicação. Alguns exemplos desses padrões incluem:

1. *WS-Reliable Messaging* é um padrão para troca de mensagens que assegura que elas serão entregues uma e somente uma vez.
2. *WS-Security* é um conjunto de padrões que dão suporte à segurança de Web services incluindo padrões que especificam a definição das políticas e padrões de proteção que cobrem o uso de assinaturas digitais.
3. *WS-Addressing* define como as informações de endereçamento devem ser representadas em uma mensagem SOAP.
4. *WS-Transactions* define como as transações por meio dos serviços distribuídos devem ser coordenadas.

Padrões de Web services são um tópico vasto e não tenho espaço para explicá-los detalhadamente aqui. Recomendo o livro do Erl (Erl, 2004; Erl, 2005) para uma visão geral desses padrões. As suas descrições detalhadas estão também disponíveis como documentos públicos na Web.

Como explico na seção seguinte, um serviço pode ser considerado simplesmente como uma abstração reusável e por isso este capítulo complementa os capítulos 18 e 19, que falam sobre problemas de reuso de software. Há, portanto, dois temas para o capítulo:

1. Engenharia de serviços. Trata do desenvolvimento de serviços confiáveis e reusáveis. Essencialmente, o assunto é o desenvolvimento de software para o reuso.
2. Desenvolvimento de software com serviços. Trata do desenvolvimento de sistemas de software confiáveis que usam serviços tanto de componentes próprios quanto em conjunto com outros tipos de componentes. Essencialmente, o assunto é o desenvolvimento de software com reuso.

Arquiteturas orientadas a serviços e engenharia de software orientada a serviços são, atualmente, 'tópicos quentes'. Há interesse do mercado em adotar a abordagem de orientação a serviços para o desenvolvimento de software, mas até o momento em que este livro estava sendo escrito, experiências práticas com sistemas orientados a serviços eram restritas. Tópicos quentes sempre geram visões ambíguas e sempre prometem mais do que finalmente fornecem. Por exemplo, no seu livro sobre SOA, Newcomer e Lomow (2005) estabelecem:

> *Dirigida pela convergência de tecnologias principais e pela adoção universal dos Web services, a empresa orientada a serviços promete incrementar significativamente a agilidade corporativa, o tempo para colocar no mercado novos produtos e serviços, reduzir os custos com TI e incrementar a eficiência operacional.*

Acredito que a engenharia de software orientada a serviços seja tão importante quanto o desenvolvimento da engenharia de software orientada a objeto. No entanto, a realidade é que ainda levará muitos anos até que se possam perceber esses benefícios e para que a visão do SOA se torne realidade. Pelo fato de o desenvolvimento de software orientado a serviço ser ainda muito recente, não temos métodos da engenharia de software bem estabelecidos para esse tipo de sistema. Portanto, foco aqui os problemas gerais de serviços de projeto e de implementação e a construção de sistemas por meio do uso de composição de serviços.

31.1 Serviços como componentes reusáveis

No Capítulo 19, apresentei a engenharia de software baseada em componentes (CBSE — *Component Based Software Engineering*), em que sistemas de software são construídos pela composição de componentes de software baseados em algum modelo de padrão de componente. Serviços são um desenvolvimento natural de componentes de software nos

quais o modelo de componente é, em essência, um conjunto de padrões associados com os Web services. Um serviço pode portanto ser definido como:

Um componente de software reusável não firmemente acoplado que engloba a funcionalidade discreta que pode ser distribuída e acessada por meio de programa. Um Web service é um serviço acessado que usa protocolos-padrão da Internet e baseados em XML.

Uma distinção importante entre um serviço e um componente de software, como definido no CBSE, é que os serviços devem ser independentes e fracamente acoplados. Devem operar sempre da mesma maneira, independentemente do seu ambiente de execução. Sua interface é do tipo 'provedor' que provê acesso às funcionalidades dos serviços. Os serviços são destinados a ser independentes e usáveis em diferentes contextos. Portanto, não devem ter uma interface do tipo 'requisito' que, em CBSE, define que os outros componentes de sistemas devam estar presentes.

Os serviços podem também ser distribuídos na Internet. Eles se comunicam pela troca de mensagens, descritas em XML, e essas mensagens são distribuídas com o uso de protocolos-padrão de transporte da Internet, como HTTP e TCP/IP. Um serviço esclarece o que ele necessita de outro serviço pela definição de seus requisitos numa mensagem e a envia para aquele serviço. O serviço receptor analisa a mensagem, trata do seu processamento e, na conclusão, envia uma mensagem para o serviço requisitante. Esse serviço então analisa a resposta para extrair as informações necessárias. Diferente dos componentes de software, serviços não 'chamam' métodos associados a outros serviços.

Para ilustrar a diferença entre a comunicação via chamadas de métodos e a comunicação via passagem de mensagem, considere a situação em que você está pedindo uma refeição em um restaurante. Quando você conversa com o garçom, está envolvido numa série de interações síncronas que definem o seu pedido. Isso é semelhante às interações de componentes num sistema de software, no qual um componente chama métodos de outros componentes. O garçom anota o seu pedido junto com o pedido de outra pessoa como você, então passa a mensagem, na qual inclui detalhes de tudo o que foi pedido, para a cozinha a fim de preparar a comida. Essencialmente, o serviço garçom é passar uma mensagem para o serviço cozinha definindo a comida a ser preparada.

Ilustrei isso na Figura 31.3, que mostra o processo síncrono de realizar pedidos, e na Figura 31.4, que mostra uma mensagem hipotética XML, que espero ser auto-explicativa, que define o pedido feito por uma mesa de três pessoas. A diferença é clara — o garçom toma o pedido como uma série de interações, sendo que cada interação define parte do pedido. No entanto, o garçom tem uma interação única com a cozinha na qual a mensagem passada define o pedido completo.

Quando você pretende usar um Web service, você tem de conhecer onde o serviço está localizado (seu URI) e os detalhes da sua interface. Eles são descritos numa descrição de serviço expressa numa linguagem baseada em XML, chamada WSDL (*Web Service Description Language*). A especificação WSDL define três pontos sobre um Web service. Ela define *o que* o serviços faz, *como* ele se comunica e *onde* encontrá-lo:

1. A parte 'o que' de um documento WSDL, chamada interface, especifica quais operações o serviço apóia e define o formato das mensagens enviadas e recebidas pelo serviço.
2. A parte 'como' do documento WSDL, chamada ligação, mapeia uma interface abstrata para um conjunto concreto de protocolos. A ligação especifica os detalhes técnicos de como se comunicar com um Web service.

Figura 31.3

Interações síncronas no pedido de uma refeição.

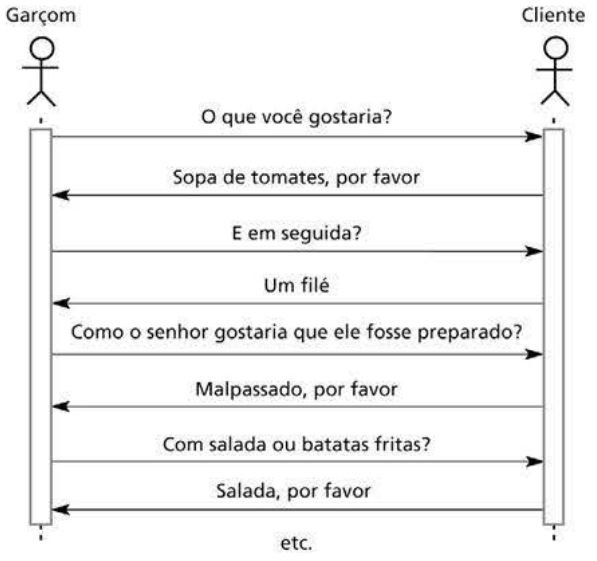


Figura 31.4

Pedido de restaurante expresso como mensagem XML.

```
<starter>
        <dish name = "soup" type = "tomato" />
        <dish name = "soup" type = "fish" />
        <dish name = "pigeon salad" />
</starter>
<main course>
        <dish name = "steak" type = "sirloin" cooking = "medium" />
        <dish name = "steak" type = "fillet" cooking = "rare" />
        <dish name = "sea bass">
</main>
<accompaniment>
        <dish name = "french fries" portions = "2" />
        <dish name = "salad" portions = "1" />
</accompaniment>
```

3. A parte 'onde' do documento WSDL, chamada (confusamente) serviço, descreve onde localizar um Web service específico implementado.

O modelo conceitual WSDL (Figura 31.5) mostra todas as partes de uma descrição de serviço. Cada uma delas é expressa em XML e pode ser fornecida em arquivos separados. Essas partes são:

1. Uma parte introdutória que, usualmente, define os namespaces XML usados e que pode incluir uma seção de documentação que forneça informações adicionais sobre o serviço.
2. Uma descrição opcional de tipos usados nas mensagens trocadas pelos serviços.
3. Uma descrição da interface de serviços, ou seja, as operações que ela fornece.
4. Uma descrição das mensagens de entrada e saída processadas pelo serviço.
5. Uma descrição da ligação usada pelo serviço, isto é, o protocolo de mensagem que será usado para enviar e receber mensagens. O default é SOAP, mas outras ligações podem ser especificadas. A ligação estabelece como as mensagens de entrada e saída associadas com o serviço devem ser empacotadas numa mensagem e especifica o protocolo de comunicação usado. A ligação pode também especificar como informações de apoio, como credenciais de proteção ou identificadores de transação, é incluída.
6. Uma especificação de ponto final que é uma localização física de um serviço, expresso como um identificador de recurso uniforme (URI — *Uniform Resource Identifier*) — endereço de um recurso que pode ser acessado por meio da Internet.

Descrições completas de serviços, escritas em XML, são extensas, detalhadas e tediosas para a leitura. Elas geralmente incluem definições de namespaces XML, que são qualificadores para nomes. Um identificador namespace pode preceder qualquer identificador usado na descrição XML. Isso significa que é possível distinguir entre identificadores com o mesmo nome que tenham sido definidos em diferentes partes de uma descrição XML. Não desejo detalhar os namespaces aqui. Para compreender este capítulo, você tem de saber somente que os nomes são prefixados com um identificador namespace e que o par **namespace:name** deve ser único.

Incluí um exemplo de uma descrição completa de serviço no site Web do livro. No entanto, como é muito longa, enfoco aqui uma descrição de interface abstrata. Essa é a parte do WSDL que se iguala à interface 'provedor' de um componente de software. A Figura 31.6 mostra detalhes da interface para um serviço simples que, dados uma data e um local (cidade ou país), retorna a máxima e a mínima temperatura registrada naquela data e local. Essas temperaturas podem ser retornadas em graus Celsius ou em graus Fahrenheit, dependendo da localização em que elas foram registradas.

Figura 31.5

Organização de uma especificação WSDL.

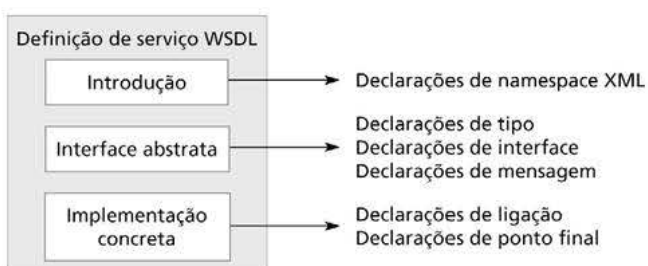


Figura 31.6

Parte de uma descrição WSDL para um serviço Web.

```
Define alguns dos tipos usados. Supõe que os prefixos namespace 'ws' referem-
se ao namespace URI dos esquemas XML e o prefixo namespace associado com
essa definição é weathns.

<types>
    <xs: schema targetNameSpace = "http://../weathns"
        xmlns: weathns = "http://.../weathns" >
        <xs:element name = "PlaceAndDate" type = "pdrec" />
        <xs:element name = "MaxMinTemp" type = "mmtrec" />
        <xs: element name = "InDataFault" type = "errmess" />

        <xs: complexType name = "pdrec"
        <xs: sequence>
            <xs:element name = "town" type = "xs:string"/>
            <xs:element name = "country" type = "xs:string"/>
            <xs:element name = "day" type = "xs:date" />
        </xs:complexType>

        Definições de MaxMinType e InDataFault aqui

    </schema>
</types>

Agora define a interface e suas operações. Nesse caso, há somente uma única
operação para retornar as temperaturas máxima e mínima

<interface name = "weatherInfo">
    <operation name = "getMaxMinTemps" pattern = "wsdlns: in-out">
    <input messageLabel = "In" element = "weathns: PlaceAndDate" />
    <output messageLabel = "Out" element = "weathns:MaxMinTemp" />
    <outfault messageLabel = "Out" element = "weathns:InDataFault" />
</operation>
</interface>
```

Na Figura 31.6, a primeira parte da descrição mostra a parte do elemento e a definição de tipo que é usado na especificação de serviço. Isso define os elementos **PlaceAndDate**, **MaxMinTemp** e **InDataFault**. Incluí somente a especificação do **PlaceAndDate**, que você pode imaginar como um registro com três campos — cidade, país e data. Uma abordagem similar poderia ser usada para definir **MaxMinTemp** e **InDataFault**.

A segunda parte da descrição mostra como a interface de serviço é definida. Nesse exemplo, o serviço **weatherInfo** tem uma única operação, embora não haja restrições no número de operações que pode ser definido. A operação **weatherInfo** tem um padrão **in-out** associado, que significa pegar uma mensagem de entrada e gerar apenas uma mensagem de saída. A especificação WSDL 2.0 permite um número de padrões diferentes de troca de mensagens, como **in-only**, **in-out**, **out-only**, **in-optional-out**, **out-in** etc. As mensagens de entrada e de saída, que se referem às definições feitas anteriormente na seção de tipos, são então definidas.

O principal problema com o WSDL é que a definição da interface de serviço não inclui quaisquer informações sobre a semântica do serviço ou suas características não funcionais, como o desempenho e a confiabilidade. É simplesmente uma descrição da assinatura do serviço e depende do usuário do serviço para deduzir o que o serviço realmente faz e o que significam os campos diferentes nas mensagens de entrada e de saída. Embora nomes significativos e a documentação de serviços auxiliem aqui, há ainda a chance para má interpretação e o mau uso do serviço.

31.2 Engenharia de serviços

Engenharia de serviços é o processo de serviços de desenvolvimento para reuso nas aplicações orientadas a serviços. Ela tem muito em comum com a engenharia de componentes. Os engenheiros de serviços têm de assegurar que o serviço representa uma abstração reusável que poderia ser útil em diferentes sistemas. Eles devem projetar e desenvolver geralmente funcionalidades úteis associadas com essas abstrações e devem assegurar que o serviço seja robusto e confiável de modo a operar confiavelmente em diferentes aplicações. Têm de documentar o serviço de modo que possa ser descoberto e compreendido por usuários potenciais.

Há três estágios lógicos no processo de engenharia de serviços (Figura 31.7). São eles:

1. Identificação do serviço candidato em que você identifica possíveis serviços que poderiam ser implementados e define os requisitos do serviço.
2. Projeto do serviço no qual você projeta a lógica e as interfaces de serviços WSDL.
3. Implementação e implantação do serviço em que você implementa e testa o serviço, e o faz disponível para o uso.

Explico cada um desses estágios nesta seção do livro.

31.2.1 Identificação de serviço candidato

A noção básica da computação orientada a serviços é que os serviços devem apoiar os processos de negócios. Como toda organização tem grande quantidade de processos, há, portanto, muitos serviços possíveis que podem ser implementados. A identificação do serviço candidato envolve compreensão e análises dos processos de negócios da organização para decidir quais serviços reusáveis são necessários para apoiar os processos.

Erl identifica três tipos fundamentais de serviço que podem ser identificados:

1. *Serviços de utilidades*. São serviços que implementam algumas funcionalidades gerais que podem ser usadas por diferentes processos de negócios. Um exemplo de um serviço de utilidade é o serviço de conversão de moeda que pode ser acessado para processar a conversão de uma (por exemplo, o dólar) para outra moeda (por exemplo, o euro).
2. *Serviços de negócios*. São serviços associados com uma função específica de negócio. Um exemplo de uma função de negócio na universidade seria o registro de estudantes para um curso.
3. *Serviços de coordenação ou de processo*. São serviços que apóiam os processos de negócio mais gerais que envolvem diferentes atores e atividades. Um exemplo de coordenação de serviços numa empresa é um serviço de pedido que permite aos pedidos serem feitos para fornecedores, mercadorias aceitas e pagamentos feitos.

Erl também sugere que os serviços possam ser considerados como orientados a tarefas ou a entidade. Os serviços orientados a tarefas são os associados com alguma atividade, uma vez que os serviços orientados a entidades são semelhantes a objetos — são associados com alguma entidade de negócio como, por exemplo, um formulário de aplicação de *job*. A Tabela 31.1 sugere alguns exemplos de serviços orientados a tarefas ou a entidades. Enquanto serviços podem ser serviços de utilidades e de negócios, os serviços de coordenação são sempre orientados a tarefas.

Seu objetivo na identificação de serviço candidato deve ser identificar serviços logicamente coerentes, independentes e reusáveis. A classificação de Erl é útil a esse respeito quando sugere como descobrir serviços reusáveis olhando as entidades e

Figura 31.7
Processo de engenharia de serviços.

Tabela 31.1 Classificação de serviços

	Utilidade	Negócio	Coordenação
Tarefa	Conversor de moeda Localizador de funcionário	Validar formulário de solicitação Verificar avaliação de crédito	Processar solicitação de reembolso Pagar fornecedor externo
Entidade	Verificador de estilo de documento Conversor de formulário Web em XML	Formulário de gastos Formulário de solicitação do estudante	

as atividades de negócios. No entanto, como os processos de identificação de objeto e de componente são difíceis, também o é a identificação de serviço candidato. Você deve pensar em possíveis candidatos e, depois, fazer uma série de perguntas sobre eles para ver se são provavelmente serviços úteis. Questões possíveis que auxiliam a identificar serviços reusáveis são:

1. Para um serviço orientado a entidade, o serviço é associado com uma única entidade lógica usada em diferentes processos de negócios? Quais são as operações normalmente executadas sobre aquela entidade que devem ser apoiadas?
2. Para serviços orientados a tarefas, a tarefa é cumprida por diferentes pessoas na organização? Elas estarão prontas a aceitar a inevitável padronização que ocorre quando um único serviço de apoio é fornecido?
3. O serviço é independente, isto é, em que extensão ele depende da disponibilidade de outros serviços?
4. Para sua operação, os serviços têm de manter o status? Se sim, um banco de dados será usado para a manutenção do status? Em geral, os sistemas que dependem dos status internos são menos reusáveis do que os em que o status pode ser mantido externamente.
5. O serviço poderia ser usado por clientes de fora da organização? Por exemplo, um serviço orientado a entidade associado com um catálogo pode ser acessado interna e externamente?
6. Espera-se que diferentes usuários do serviço tenham diferentes requisitos não funcionais? Se sim, isso sugere que mais de uma versão de um serviço deva talvez ser implementada.

As respostas a essas questões auxiliam a selecionar e refinar as abstrações que podem ser implementadas como serviços. No entanto, não há fórmulas para decisão de quais são os melhores serviços e, portanto, a identificação de serviço é uma habilidade e um processo baseados em experiência.

A saída do processo de seleção de candidatos é um conjunto de serviços identificados e requisitos associados a esses serviços. Os requisitos funcionais de serviços devem definir o que o serviço deve fazer. Os requisitos não funcionais devem definir os requisitos de proteção, desempenho e disponibilidade do serviço.

Considere que uma grande empresa, que vende equipamentos computacionais, organizou preços especiais para configurações aprovadas para alguns clientes. Para facilitar a colocação de pedidos automáticos, a empresa deseja produzir um serviço de catálogo que permitirá aos clientes selecionar o equipamento de que eles necessitam. Diferente de um catálogo de consumidor, no entanto, os pedidos não são colocados diretamente, por meio de uma interface de catálogo, mas são realizados por meio de sistemas de aquisição baseados na Web de cada empresa. A maioria das empresas possui seus próprios procedimentos de orçamento e aprovação para os pedidos e os seus próprios processos de pedidos devem ser seguidos quando um pedido é colocado.

O catálogo de serviços é um exemplo de um serviço orientado a entidade que apóia as operações de negócios. Os requisitos funcionais de serviços de catálogo funcional são:

1. Uma versão específica do catálogo deverá ser fornecida para cada empresa usuária. Isso incluirá as configurações e o equipamento que pode ser solicitado pelos empregados da empresa cliente e os preços acordados para os itens do catálogo.
2. O catálogo deverá permitir que um empregador cliente baixe uma versão do catálogo para que ele possa navegar off-line.
3. O catálogo deverá permitir que os usuários comparem as especificações e os preços até seis itens do catálogo.
4. O catálogo deverá fornecer recursos de navegação e pesquisa para os usuários.
5. Os usuários do catálogo deverão ser capazes de descobrir a data da entrega prevista para dada quantidade de itens específicos do catálogo.
6. Os usuários do catálogo deverão ser capazes de enviar 'pedidos virtuais', sendo que os itens necessários estarão reservados para eles por 48 horas. Os pedidos virtuais devem ser confirmados por um pedido real colocado por meio de um sistema de aquisição. O pedido deve ser recebido dentro de 48 horas a partir da entrada do pedido virtual.

Adicionalmente a esses requisitos funcionais, o catálogo tem uma quantidade de requisitos não funcionais:

1. O acesso ao serviço de catálogo deverá ser restrito aos empregados das organizações credenciadas.
2. Os preços e as configurações oferecidos para um cliente deverão ser confidenciais e não deverão estar disponíveis para os empregados de qualquer outro cliente.
3. O catálogo deverá estar disponível sem interrupção de serviço de 0700 GMT a 1100 GMT.
4. O serviço de catálogo deverá ser capaz de processar até 10 solicitações por segundo no pico de trabalho.

Observe que não há requisito não funcional relacionado ao tempo de resposta do serviço de catálogo. Isso depende do tamanho do catálogo e do número esperado de usuários simultâneos. Como não é um serviço de tempo crítico, não há necessidade de especificá-lo nesse estágio.

31.2.2 Projeto de interface de serviço

Uma vez que você selecionou os serviços candidatos, o próximo estágio no processo da engenharia de serviço é projetar as interfaces de serviços. Isso envolve a definição de operações associadas com o serviço e seus parâmetros. Você também deve pensar cuidadosamente sobre como as operações e as mensagens de serviço podem ser projetadas para minimizar o número de troca de mensagens que deve ocorrer para completar a solicitação de serviço. Você tem de assegurar que o maior o número possível de informações é passado para o serviço em uma mensagem antes de requerer interações síncronas de serviço.

Você deve também lembrar que os serviços são sem *status* e o gerenciamento de um status de aplicação específica de serviço é de responsabilidade do usuário do serviço em vez do serviço em si. Você portanto tem que passar essas informações de status para os dos serviços nas mensagens de entrada e de saída.

Há três estágios para o projeto de interface de serviço:

1. Projeto de interface lógica no qual você identifica as operações associadas com o serviço, as entradas e saídas dessas operações e as exceções associadas com essas operações.
2. Projeto de mensagem no qual você projeta a estrutura das mensagens enviadas e recebidas pelo serviço.
3. Desenvolvimento WSDL no qual você traduz seu projeto lógico e de mensagem para uma descrição abstrata de interface escrita em WSDL.

O primeiro estágio, projeto lógico de interface, inicia com os requisitos de serviços e define os nomes das operações e parâmetros associados com o serviço. Nesse estágio, você deve também definir as exceções que podem aparecer quando uma operação de serviço for invocada. As tabelas 31.2 e 31.3 mostram as operações que implementam os requisitos e as entradas, saídas e exceções para cada uma das operações do catálogo. Nesse estágio, não há necessidade de as operações serem especificadas em detalhes — você adiciona os detalhes no próximo estágio do processo de projeto.

A definição de exceções e como elas podem ser comunicadas aos usuários do serviço é particularmente importante. Os engenheiros de serviços desconhecem como seus serviços serão usados e normalmente é imprudente formular hipóteses de que os usuários do serviço terão uma compreensão total da especificação do serviço. As mensagens de entradas podem ser incorretas, portanto você deve definir exceções que relatem entradas incorretas para o cliente do serviço. É geralmente uma boa prática em desenvolvimento de componentes reusáveis deixar todo tratamento de exceções para o usuário do componente — o desenvolvedor do serviço não deve impor sua visão de como as exceções devem ser tratadas.

Uma vez que você estabeleceu uma descrição lógica informal do que o serviço deve fazer, o próximo estágio é definir a estrutura das mensagens de entrada e de saída e os tipos usados nessas mensagens. O XML é uma notação inadequada para se usar nesse estágio. Penso que é melhor representar as mensagens como objetos e defini-las usando o UML ou uma linguagem de programação, como Java. Elas podem então ser convertidas manual ou automaticamente para o XML. A Figura 31.8 é um diagrama UML, que mostra a estrutura de mensagens de entrada e de saída para a operação **getDelivery** no serviço de catálogo.

Perceba como adicionei detalhes à descrição, pelas anotações no diagrama UML com restrições. Elas definem o comprimento das strings que representam a empresa e o item de catálogo, especificam que o número de itens deve ser maior do que zero e que a entrega deve ser depois da data atual. As anotações mostram também quais códigos de erros são associados com cada defeito possível.

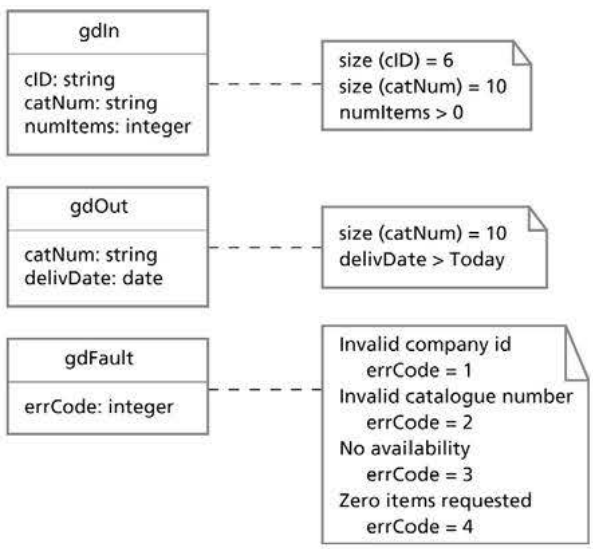
Tabela 31.2 Descrições funcionais de operações de serviços de catálogo

Operação	Descrição
MakeCatalogue	Cria uma versão do catálogo feita sob medida para um cliente específico. Inclui um parâmetro opcional para criar uma versão do catálogo em PDF que possa ser baixada.
Compare	Fornece uma comparação de até seis características (por exemplo preço, dimensões, velocidade do processador etc.), de até quatro itens de catálogo por vez.
Lookup	Mostra todos os dados associados a um item específico do catálogo.
Search	Essa operação busca uma expressão lógica no catálogo. Ela mostra uma lista de todos os itens que possuem a expressão procurada.
CheckDelivery	Retorna a data de entrega prevista de um item pedido hoje.
MakeVirtualOrder	Reserva o número de itens a serem pedidos por um cliente e fornece informações do item para o sistema de compras dos clientes.

Tabela 31.3 Projeto de interface de catálogo

Operação	Entradas	Saídas	Exceções
MakeCatalogue	*mcIn* ID da empresa PDF-flag	*mcOut* URL do catálogo para essa empresa	*mcFault* ID da empresa inválido
Compare	*compIn* ID da empresa Atributo de entrada (até 6) Número de catálogo (até 4)	*compOut* URL da página que mostra a tabela de comparação	*compFaul* ID da empresa inválido Número de catálogo inválido Atributo desconhecido
Lookup	*lookIn* ID da empresa Número de catálogo	*lookOut* URL da página com as informações do item	*lookFault* ID da empresa inválido Número de catálogo inválido
Search	*searchIn* Id da empresa String de busca	*searchOut* URL da página com resultados da busca	*searchFault* ID da empresa inválido String de busca mal-estruturado
CheckDelivery	*gdIn* ID da empresa Número de catálogo Número de itens solicitados	*gdOut* Número de catálogo Data de entrega prevista	*gdFault* ID da empresa inválido Número de catálogo inválido Sem disponibilidade
PlaceOrder	*poIn* ID da empresa Número de itens solicitados Número de catálogo	*poOut* Número de catálogo Número de itens solicitados Data de entrega prevista Estimativa de preço unitário Estimativa de preço total	*poFault* ID da empresa inválido Número de catálogo inválido Zero itens solicitados

Figura 31.8
Definição UML de mensagens de entrada e saída.



O estágio final do processo de projeto de serviço é traduzir o projeto de interface de serviço em WSDL. Como expliquei na seção anterior, uma representação WSDL é longa e detalhada, e por isso é fácil cometer enganos nesse estágio. A maioria dos ambientes de programação que apóiam desenvolvimento orientado a serviços (por exemplo, o ambiente ECLIPSE) inclui ferramentas que podem traduzir uma descrição lógica de interface na sua correspondente representação WSDL.

31.2.3 Implementação e implantação de serviço

Uma vez que você identificou os serviços candidatos e projetou as interfaces, o estágio final do processo de engenharia de serviços é a implementação do serviço. Essa implementação pode envolver a programação dos serviços usando uma linguagem padronizada de programação, como Java ou C#. Ambas as linguagens atualmente incluem bibliotecas com apoio extensivo para desenvolvimento de serviços.

De outro modo, os serviços podem ser desenvolvidos com o uso de componentes existentes ou, como explico a seguir, sistemas legados. Isso significa que os ativos de software que já provaram a sua utilidade podem estar mais largamente disponíveis. No caso de sistemas legados, isso pode significar que a funcionalidade de sistema pode ser acessada pelas novas aplicações. Novos serviços podem, também, ser desenvolvidos por meio da definição de composições de serviços existentes. Explico o desenvolvimento pela composição de serviço na Seção 31.3.

Na medida em que o serviço tenha sido implementado, ele deve então ser testado antes que seja implantado. Isso envolve o exame e o particionamento das entradas de serviços (como explicado no Capítulo 23), a criação de mensagens de entrada que refletem essas combinações de entradas e, então, a verificação das saídas esperadas. Você deve sempre tentar gerar exceções durante o teste para verificar se o serviço pode lidar com entradas inválidas. Estão atualmente disponíveis diversas ferramentas de teste que permitem aos serviços serem examinados e testados e que geram os testes com base em uma especificação WSDL. Contudo, eles podem somente testar a conformidade da interface de serviço ao WSDL. Não podem testar se o comportamento funcional do serviço está conforme o especificado.

A implantação do serviço, o estágio final do processo, envolve tornar o serviço disponível para o uso no servidor Web. Muitos softwares de servidores tornam isso muito simples. Você tem somente de instalar o arquivo que contém o serviço executável num diretório específico. Então, torna-se automaticamente disponível para o uso. Se a intenção for disponibilizar o serviço publicamente, você tem de escrever uma descrição UDDI para que usuários potenciais possam achar o serviço. Erl (2004) fornece um resumo útil do UDDI em seu livro.

Há atualmente um número de registros públicos para descrições UDDI e os negócios podem também manter o registro privado de UDDI. Uma descrição UDDI consiste em um número de diferentes tipos de informações:

1. Detalhes dos negócios que fornecem o serviço. Isso é importante por motivos de confiabilidade. Usuários de um serviço têm de ser confiáveis no sentido de que não terão comportamento malicioso. As informações sobre o provedor dos serviços permitem aos usuários verificarem as credenciais do provedor.
2. Descrição informal das funcionalidades fornecidas pelo serviço. Auxilia os usuários potenciais a decidirem se o serviço é o que eles desejam. No entanto, a descrição funcional é em linguagem natural, portanto ela não é uma descrição semântica e sem ambigüidade do que o serviço faz.
3. Informações sobre onde encontrar a especificação WSDL associada ao serviço.
4. Informações de assinatura que permitem aos usuários se registrarem para obter informações sobre atualizações de serviços.

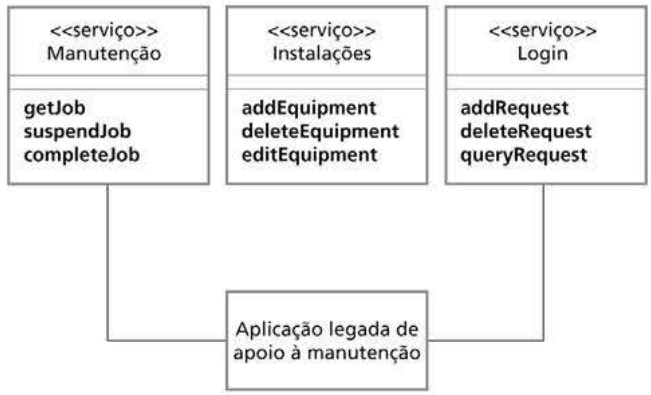
Um problema potencial com especificações UDDI é que o comportamento funcional dos serviços é especificado informalmente como uma descrição em linguagem natural. Conforme expliquei no Capítulo 6, que aborda requisitos de software, descrições em linguagem natural são fáceis de ler, mas são sujeitas à má interpretação. Para resolver esse problema, há uma comunidade ativa de pesquisa destinada à investigação de como a semântica de serviços pode ser especificada. A abordagem mais promissora para especificação de semântica é a descrição baseada em ontologia. Uma linguagem chamada OWL-S foi desenvolvida para descrever ontologias de Web service (OWL_Services_Coalition, 2003). Até o momento em que este livro estava sendo escrito, essas técnicas para especificação de semântica de serviços eram ainda imaturas, porém se tornarão largamente usadas nos próximos anos.

31.2.4 Serviços de sistemas legados

No Capítulo 18, expliquei a possibilidade de implementação de componentes reusáveis pelo fornecimento de uma interface de componente para sistemas legados existentes. Em essência, as funcionalidades dos sistemas legados podem ser reusáveis. A implementação do componente foi simplesmente destinada a fornecer uma interface geral para esse sistema. Um dos usos mais importantes de serviços é implementar 'wrappers' em sistemas legados. Esses sistemas podem então ser acessados por meio da Web e integrados com outras aplicações.

Para ilustrar isso, imagine que uma grande empresa mantém um inventário sobre seus equipamentos e um banco de dados de manutenção associado. Mantém-se a rastreabilidade sobre quais solicitações de manutenção foram feitas em diferentes peças de equipamentos, com que regularidade a manutenção é programada, quando a manutenção foi realizada, quanto tempo foi gasto na manutenção etc. Esse sistema legado foi originalmente usado para gerar diariamente listas de tarefas para a equipe

Figura 31.9
Serviços que fornecem acesso a um sistema legado.



de manutenção, mas, durante esse tempo, novos recursos foram adicionados. Eles fornecem dados sobre quanto foi gasto na manutenção de cada peça de equipamento e informações para auxiliar a estabelecer o custo do trabalho de manutenção a ser realizado por empresas externas. O sistema é executado como um sistema cliente–servidor com software de cliente de propósito especial que roda num PC.

A empresa agora deseja fornecer acesso de tempo real no sistema com base em terminais portáteis usados pela equipe de manutenção. Eles atualizarão o sistema diretamente com o tempo e os recursos gastos na manutenção e consultarão o sistema para encontrar sua próxima tarefa de manutenção. Adicionalmente, a equipe do centro de chamadas solicitará acesso no sistema para registrar solicitações de manutenção e verificar o seu status.

É praticamente impossível aprimorar o sistema para apoiar esses requisitos; assim, a empresa decide fornecer novas aplicações para as equipes de manutenção e de centro de chamadas. Essas aplicações dependem do sistema legado usado como base para implementar uma quantidade de serviços. Isso é ilustrado na Figura 31.9, na qual usei um estereótipo da UML para indicar um serviço. Novas aplicações simplesmente trocam mensagens com esses serviços para acessar as funcionalidades do sistema legado.

Alguns dos serviços oferecidos são:

1. *Serviço de manutenção.* Inclui operações para recuperar uma tarefa de manutenção de acordo com o número da tarefa, a prioridade e a localização geográfica e carregar detalhes da manutenção que tenha sido realizada no banco de dados de manutenção. Ele apóia também uma operação para permitir que a manutenção iniciada, mas incompleta, seja suspensa.
2. *Serviço de recursos.* Inclui operações para adicionar e excluir novos equipamentos e modificar as informações associadas ao equipamento no banco de dados.
3. *Serviço de log.* Inclui operações para adicionar uma nova solicitação para um serviço, excluir solicitações de manutenção e consultar o status das solicitações pendentes.

A existência de sistema legado não é representada simplesmente como um serviço único. Pelo contrário, os serviços desenvolvidos são coerentes e apóiam uma única área de funcionalidade. Isso reduz sua complexidade e os torna mais fáceis de entender e de reusar em outras aplicações. Não tenho espaço para explicar os detalhes das mensagens que poderiam ser trocadas por esses serviços — o projeto é deixado como um exercício para o leitor.

31.3 Desenvolvimento de software como serviços

O desenvolvimento de software usando serviços baseia-se na idéia de que você compõe e configura serviços para criar um novo serviço composto. Estes podem ser integrados a uma interface com o usuário Web para criar uma aplicação Web ou ser usados como componentes em alguma outra composição de serviços. Os serviços envolvidos numa composição podem ser especialmente desenvolvidos para uma aplicação, podem ser serviços de negócios desenvolvidos dentro de uma empresa ou podem ser serviços para algum provedor externo.

Muitas empresas dedicam-se à conversão de aplicações usadas dentro de uma companhia em sistemas orientados a serviços. Abre-se a possibilidade para ampliar o reuso dentro da empresa. O próximo estágio será o desenvolvimento de aplicações interorganizacionais entre fornecedores confiáveis. A realização final da visão de longo prazo das arquiteturas orientadas a serviços será contar com o desenvolvimento de um 'mercado de serviços'. Penso que é improvável que isso venha à tona durante o tempo de vida deste livro. Enquanto o livro estava sendo escrito, somente um número relativamente pequeno de serviços de negócios que poderiam ser incluídos em aplicações estavam publicamente disponíveis.

A composição de serviço pode ser usada para integrar processos de negócios separados a fim de fornecer um processo integrado que ofereça funcionalidades mais extensas. Digamos que uma empresa aérea deseje oferecer um pacote completo de férias para os viajantes. Assim como fazem as suas reservas de vôos, os viajantes podem também fazer reservas em hotéis na sua localização preferida, acertar o aluguel de carro ou reservar um táxi do aeroporto, pesquisar um guia de viagem e fazer reservas para visitar atrações locais. Para criar essa aplicação, a empresa aérea compôs o seu próprio serviço de reserva com os serviços oferecidos por uma agência de reservas de hotel, empresas de aluguel de carro e de táxi e serviços de reserva oferecidos pelos provedores de atrações locais. O resultado é um serviço único que integra esses serviços de provedores diferentes.

Você pode pensar nesse processo como uma seqüência de passos separados como mostrado na Figura 31.10. As informações são passadas de um passo para o seguinte — por exemplo, a empresa de aluguel de carro é informada sobre a hora que o vôo é programado para chegar. A seqüência dos passos é chamada de workflow — conjunto de atividades ordenadas no tempo cuja cada atividade realiza alguma parte do trabalho. Você pode pensar no workflow como um modelo de processo de negócio — os passos envolvidos para se atingir algum objetivo importante para um negócio. Nesse caso, o processo de negócio é o serviço de reservas de férias oferecido pela empresa aérea.

O workflow é uma idéia simples e o cenário anterior de reserva de férias parece ser muito direto. Na realidade, a composição de serviços é muito mais complexa do que implica esse simples modelo. Por exemplo, você tem de considerar a possibilidade de uma falha de serviço e incorporar mecanismos para tratar essas falhas. Você tem de levar em consideração as demandas excepcionais feitas pelos usuários da aplicação. Por exemplo, digamos que um viajante era inválido e solicitou o aluguel de uma cadeira de rodas para ser entregue no aeroporto.

Você deve ser capaz de lidar com situações em que o workflow precise ser alterado porque a execução normal de um dos serviços resulta em incompatibilidade com alguma outra execução de serviço. Por exemplo, digamos que um vôo está programado para partir no dia 1º de junho e retornar no dia 7 de junho. O wokflow, então, prossegue para o estágio de reserva no hotel. No entanto, o resort está hospedando uma grande conferência até o dia 2 de junho; portanto, os quartos não estão disponíveis. O serviço de reservas do hotel reporta essa falta de disponibilidade. Não é uma falha: a falta de disponibilidade é uma situação comum. Você então tem de desfazer a reserva de vôo e transferir as informações sobre a falta de disponibilidade de volta para o usuário. Ele ou ela tem de decidir se troca as suas datas ou de resort. Na terminologia do workflow, uma 'ação de compensação' é usada para desfazer ações que já foram completadas.

O processo de projeto de um novo serviço pela composição de serviços existentes é, essencialmente, um processo de projeto de software com reuso (Figura 31.11). O projeto com reuso envolve inevitavelmente compromissos com requisitos. Os requisitos 'ideais' para o sistema têm de ser modificados para refletir os serviços que estão realmente disponíveis, cujos custos estão dentro do orçamento e cuja qualidade de serviços é aceitável.

Na Figura 31.11, mostrei seis estágios principais no processo de construção de serviço pela composição:

1. *Formular as principais características de um workflow*. Nesse estágio inicial do projeto de serviço, você usa os requisitos para o serviço composto como uma base para criar o projeto 'ideal' do serviço. Você deve criar um projeto totalmente abstrato nesse estágio com a intenção de adicionar detalhes uma vez que você conhece mais sobre os serviços disponíveis.

Figura 31.10 Workflow de pacote de férias.

Figura 31.11 Construção de serviços por composição.

2. *Descobrir serviços.* Durante esse estágio do processo, você procura registros de serviço para descobrir quais serviços existem, quem os fornece e os detalhes do fornecimento de serviço.
3. *Selecionar serviços possíveis.* Do conjunto de serviços candidatos possíveis que você descobriu, você então seleciona os serviços possíveis que possam implementar as atividades de workflow. Seu critério de seleção incluirá, obviamente, a funcionalidade dos serviços oferecidos. Ele também pode incluir o custo de serviços e a qualidade de serviço (tempo de resposta, disponibilidade etc.) oferecidos. Você pode decidir escolher um número de serviços funcionalmente equivalente, que pode ser limitado a uma atividade de workflow dependendo dos detalhes de custo e de qualidade de serviço.
4. *Refinar o workflow.* Com base nas informações sobre os serviços que você selecionou, você deve então refinar o workflow. Isso envolve a adição de detalhes para uma descrição abstrata e, talvez, adição e remoção de atividades de workflow. Pode-se depois repetir os estágios de descoberta e seleção de serviço. Uma vez que um conjunto de serviços estáveis foi escolhido e o projeto final do workflow foi estabelecido, passa-se para o próximo estágio do processo.
5. *Criar o programa de workflow.* Durante esse estágio, o projeto abstrato de workflow é transformado num programa executável e a interface de serviço é definida. Você pode usar uma linguagem de programação convencional, como Java ou C#, para implementação de serviço ou pode usar uma linguagem mais especializada de workflow como WS-BPEL. Conforme expliquei na seção anterior, a especificação de interface de serviço deve ser escrita em WSDL. Esse estágio pode também envolver a criação de interfaces com o usuário baseadas em Web para permitir que o novo serviço seja acessado de um navegador Web.
6. *Testar o serviço ou a aplicação completada.* O processo de teste de serviço composto e completo é mais complexo do que o teste de componente em situações em que os serviços externos são usados. Explico questões de teste na Seção 31.3.2.

No fim deste capítulo, foco o projeto e o teste de workflow. Como expliquei na introdução, um mercado para serviços ainda não foi desenvolvido. Embora vários registros públicos UDDI estejam disponíveis, eles não são muito completos e as descrições de serviço são algumas vezes vagas e insuficientes. Por essas razões, a descoberta de serviços não é ainda uma das principais questões. A maioria dos serviços será descoberta dentro das organizações onde os serviços possam ser achados com a utilização de registros internos e comunicações informais entre os engenheiros de software.

31.3.1 Projeto e implementação de workflow

O projeto de um workflow envolve a análise de processos existentes ou planejados de negócio para compreender os diferentes estágios desses processos e então poder representar os que estão sendo projetados numa notação de projeto de workflow. Isso mostra os estágios de aprovação dos processos e das informações passadas entre diferentes estágios dos processos. No entanto, o processo atual pode ser informal e dependente das capacidades e habilidades das pessoas envolvidas — pode não haver um meio 'normal' de trabalho. Nesses casos, você tem de usar o conhecimento de processo para projetar um workflow que alcance os mesmos objetivos dos processos atuais de negócio.

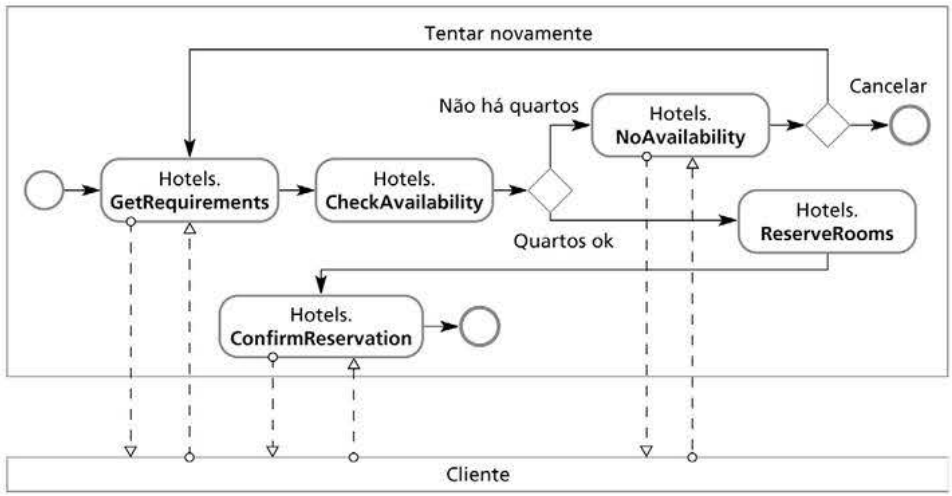
Os workflows representam modelos do processo de negócio e são normalmente representados por meio de uma notação gráfica, como BPMN (White, 2004) ou YAWL (Van der Aalst e ter Hofstede, 2005). Até o momento em que este livro estava sendo escrito, a linguagem de modelagem de processo que parece emergir como padrão é a BPMN. É uma linguagem gráfica razoavelmente fácil de entender. Os mapeamentos foram definidos para traduzir a linguagem para baixo nível, descrições baseadas em XML em WS-BPEL. BPMN está, portanto, em conformidade com a pilha de padrões de Web services que mostrei na Figura 31.2. Uso a notação BPMN aqui para ilustrar a noção da programação de processo de negócio.

A Figura 31.12 é um exemplo de um modelo simples de BPMN de uma parte do cenário de pacote de férias anterior. O modelo mostra um workflow simplificado de reserva de hotel e considera a existência de um serviço Hotels com operações associadas chamadas de **GetRequirements**, **CheckAvailability**, **ReserveRooms**, **NoAvailability**, **ConfirmReservation** e **CancelReservation**. Os processos envolvem a obtenção de requisitos do cliente, verificação da disponibilidade de quartos e, se estiverem disponíveis, efetuação da reserva para a data solicitada.

Esse modelo introduz alguns dos principais conceitos do BPMN usados para criar modelos de workflow:

1. Atividades são representadas por um retângulo com cantos arredondados. Uma atividade pode ser executada por uma pessoa ou por um serviço automatizado.
2. Eventos são representados por círculos. Um evento é algo que acontece durante um processo de negócio. Um círculo simples é usado para representar um evento inicial e um círculo mais escuro para representar um evento final. Um círculo duplo (não mostrado) é usado para representar um evento intermediário. Os eventos podem ser eventos de relógios, permitindo assim aos workflows serem executados periodicamente ou interrompidos.

Figura 31.12 Workflow de reserva de hotel.



3. Um losango é usado para representar um *gateway*. Um gateway é um estágio no processo em que alguma escolha é feita. Por exemplo, na Figura 31.12, há uma escolha feita com base na disponibilidade ou não dos quartos.
4. Uma seta sólida é usada para mostrar a seqüência de atividades; uma seta tracejada representa um fluxo de mensagem entre atividades — na Figura 31.12, essas mensagens são passadas entre o serviço de reserva de hotel e o cliente.

Essas características principais são suficientes para descrever a essência da maioria dos workflows. No entanto, o BPMN inclui diversas características adicionais, mais não tenho espaço para descrevê-las aqui. Essas informações adicionadas à descrição de um processo de negócio permitem que ele seja automaticamente traduzido em uma forma executável. Portanto, o Web service, baseado em composições de serviços descritos em BPMN, pode ser criado de um modelo de processo de negócio.

A Figura 31.12 mostra o processo executado em uma organização — a empresa que fornece um serviço de reservas. No entanto, o benefício principal de uma abordagem orientada a serviços é que ele apóia a computação interorganizacional. Isso significa que a computação total envolve serviços em diferentes empresas. Tal fato é representado em BPMN pelo desenvolvimento separado de workflows para cada uma das organizações envolvidas com as suas interações.

Como ilustração, uso um exemplo diferente, tirado de *grid computing*. Uma abordagem orientada a serviços foi proposta para permitir que recursos, como computadores de alto desempenho, sejam compartilhados. Nesse exemplo, suponha que um computador de processamento vetorial (uma máquina que pode realizar computações paralelas em vetores de valores) seja oferecido como um serviço (**VectorProcService**) por um laboratório de pesquisa. Ele é acessado por meio de um outro serviço chamado **SetupComputation**. Os serviços e as interações são mostrados na Figura 31.13.

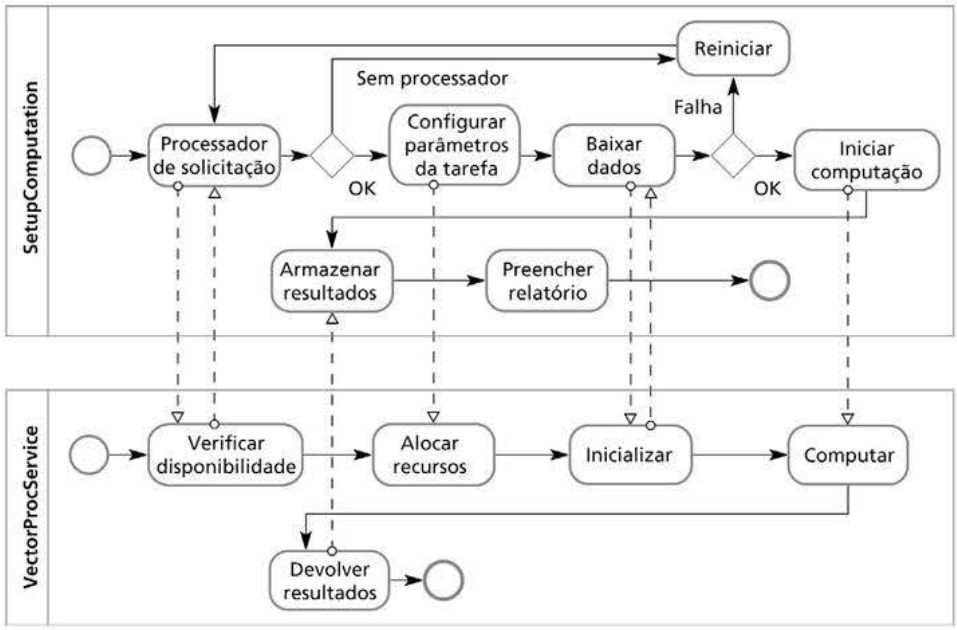
Nesse exemplo, o workflow para as solicitações de serviço **SetupComputation** acessa um processador vetorial e, se um processador estiver disponível, estabelece a computação e baixa os dados para o serviço de processamento. Uma vez completada a computação, os resultados são armazenados no computador local. O workflow para **VectorProcService** verifica se um processador está disponível, aloca recursos para a computação, inicia o sistema, realiza a computação e devolve os resultados para o serviço de cliente.

Em termos de BPMN, o workflow de cada organização é representado num pool separado. Ele é mostrado graficamente pelo enquadramento do workflow para cada participante do processo, num retângulo, com seu nome escrito verticalmente do lado esquerdo. Os workflows definidos em cada pool são coordenados pela troca de mensagens; o fluxo seqüencial entre as atividades em diferentes pools não é permitida. Em situações em que diferentes partes de uma organização estão envolvidas num workflow, podem ser mostradas pela separação dos pools em 'raias'. Cada raia mostra as atividades naquela parte da organização.

Na medida em que um modelo de processo de negócio foi projetado, ele tem de ser refinado dependendo dos serviços que foram descobertos. Conforme sugeri na explicação da Figura 31.11, o modelo caminha através de um número de interações até que um projeto, que permita o máximo possível de reuso de serviços disponíveis, seja criado. Uma vez que um projeto esteja disponível, o próximo passo é convertê-lo num programa executável. Como os serviços são independentes de linguagem de implementação, podem ser escritos em qualquer linguagem e os ambientes de desenvolvimento Java e C# fornecem apoio para a composição de Web service.

Para fornecer apoio direto para a implementação das composições de Web service, muitos padrões de Web service foram desenvolvidos. O mais conhecido é a linguagem de execução de processos de negócio (WS-BPEL — *Business*

Figura 31.13
Interação de workflows.



Process Execution Language), que é uma linguagem de programação baseada em XML para controlar as interações de serviços. Ela é apoiada pelos padrões adicionais, como WS-Coordination (Cabrera, et al., 2005), o qual é usado para especificar como os serviços são coordenados e a linguagem de descrição de coreografia (WS-CDL — *Choreography Description Language*) (Kavantzas, et al., 2004), que é um meio de definição das trocas de mensagens entre os participantes (Andrews, et al., 2003).

Todos eles são padrões XML de modo que as descrições resultantes são longas e difíceis de ler. Escrever programas diretamente em notações baseadas em XML é um processo lento e propenso a erros. Decidi portanto não caminhar para os detalhes das notações baseadas em XML, como WS-BPEL, porque não são essenciais para compreender os princípios do workflow e composição de serviços. Como suporte para a maturação da computação orientada a serviços, essas descrições XML serão geradas automaticamente. As ferramentas analisarão gramaticalmente uma descrição gráfica de workflow e gerarão composições de serviços executáveis.

31.3.2 Teste de serviços

A realização de testes é importante em todos os processos de desenvolvimento de sistemas para auxiliar a demonstrar que o sistema atende a seus requisitos funcionais e não funcionais e para detectar defeitos que foram introduzidos durante o processo de desenvolvimento. Conforme expliquei nos capítulos 22 a 24, uma gama de abordagens diferentes para a validação de sistemas e testes foi desenvolvida para dar suporte ao processo de testes. Muitas dessas técnicas, como as inspeções de programas e o teste de cobertura, destinam-se à análise do código-fonte de software. No entanto, quando os serviços são oferecidos por um provedor externo, o código-fonte da implementação de serviços não está disponível. Os testes baseados em serviços não podem portanto usar técnicas aprovadas baseadas em código-fonte.

Assim como os problemas para compreender a operação de serviço, os testadores podem se defrontar com dificuldades adicionais quando testam serviços e composições de serviços:

1. Serviços externos estão sob o controle do provedor de serviços, e não dos seus usuários. O provedor de serviços pode retirar esses serviços na hora em que desejar alterá-los, o que invalida qualquer experiência anterior de teste. Esses problemas são tratados nos componentes de software mantendo-se as diferentes versões. Atualmente, no entanto, não há padrões propostos para lidar com versões de serviços.
2. A visão de longo prazo para as arquiteturas orientadas a serviços é os serviços serem limitados dinamicamente pelas aplicações orientadas a serviços. Isso significa que uma aplicação pode nem sempre usar o mesmo serviço cada vez que ele é executado. Portanto, os testes podem ser bem-sucedidos quando uma aplicação for limitada por algum serviço específico, mas não podem garantir que o serviço seja usado durante a execução real do sistema.
3. Como, na maioria dos casos, um serviço está disponível para diferentes clientes, o comportamento não funcional desse serviço não depende simplesmente de como ele é usado por uma aplicação que está sendo testada. Um serviço

pode executar bem durante o teste porque ele não está operando sob grande carga de trabalho. Na prática, o comportamento de serviço observado pode ser diferente devido às demandas feitas pelos outros usuários.

4. O modelo de pagamento para os serviços pode tornar os testes muito caros. Há diferentes modelos possíveis de pagamento — alguns serviços podem estar disponíveis sem custo, alguns são pagos por assinatura e outros pagos conforme o uso. Se os serviços são grátis, o provedor de serviço não desejará tê-los carregados por aplicações em teste; se uma assinatura for necessária, um usuário de serviço pode relutar em concordar com um acordo de assinatura antes de testar o serviço; se o uso baseia-se no pagamento por cada uso, os usuários de serviços podem achar proibitivo o custo dos testes.
5. Expliquei a noção de ações de compensação invocadas quando alguma exceção ocorre e os compromissos anteriores que foram feitos (como uma reserva do vôo) têm de ser revogados. Há um problema em testar essas ações quando elas podem depender de falhas de outros serviços. Afirmar que esses serviços realmente falhem durante um processo de teste pode ser muito difícil.

Esses problemas são particularmente agudos quando são usados serviços externos. São menos sérios quando os serviços são usados dentro de uma mesma empresa ou quando as empresas parceiras confiam nos serviços oferecidos pelos seus parceiros. Nesses casos, o código-fonte pode estar disponível para guiar o processo de teste, e o pagamento pelo serviço não parece ser um problema. A solução desses problemas de teste e a produção de guias, ferramentas e técnicas para testes de aplicações orientadas a serviços é atualmente um campo importante de pesquisa.

PONTOS-CHAVE

- A engenharia de software orientada a serviços baseia-se na noção de que os programas podem ser construídos pela composição de serviços independentes que englobam funcionalidade reusável. Os serviços são independentes de linguagem e sua implementação fundamenta-se em padrões amplamente adotados baseados em XML.
- Interfaces de serviços são definidas em uma linguagem baseada em XML chamada WSDL. Uma especificação WSDL inclui uma definição de tipos de interface e operações, o protocolo de ligação usado pelo serviço e a localização do serviço.
- Serviços podem ser classificados como serviços de utilidades que fornecem alguma funcionalidade de propósito geral, serviços de negócios que implementam parte de um processo de negócio ou serviços de coordenação que coordenam a execução de outros serviços.
- O processo da engenharia de serviços envolve a identificação de serviços candidatos para implementação, definição de interface de serviços e implementação, teste e implantação de serviços.
- Interfaces de serviço podem ser definidas para sistemas legados de software que continuem sendo úteis para uma organização. A funcionalidade do sistema legado pode, portanto, ser reusada em outras aplicações.
- O desenvolvimento do software que usa serviços baseia-se na idéia de que os programas são criados pela composição e configuração de serviços para criar novos serviços compostos.
- Modelos de processo de negócio definem as atividades e troca de informações que ocorrem em algum processo de negócio. As atividades em um processo de negócio podem ser implementadas por serviços e, desse modo, o modelo de processo de negócio representa uma composição de serviço.
- Técnicas de teste de software baseadas na análise de código-fonte não podem ser usadas em sistemas orientados a serviços que contam com serviços fornecidos externamente.

LEITURAS SUGERIDAS

Há uma imensa quantidade de material do tipo tutorial na Web que abrange todos os aspectos dos Web services. Contudo, encontrei os dois livros de Thomas Erl, que considero as melhores visões gerais e descrição de serviços e padrões de serviços. Diferentemente de muitos livros, Erl inclui uma explicação dos problemas de engenharia de software na computação orientada a serviços.

Erl, T. (2004). *Service-oriented architecture: a field guide to integrating XML and Web services*, Upper Saddle River, NJ: Prentice-Hall.

Erl, T. (2005). *Service-oriented architecture: concepts, technology and design,* Upper Saddle River, NJ: Prentice-Hall.

EXERCÍCIOS

31.1 Quais são as distinções importantes entre serviços e componentes de software?

31.2 Explique por que as arquiteturas orientadas a serviços devem basear-se em padrões.

31.3 Por que é importante minimizar o número de mensagens trocadas pelos serviços?

31.4 Explique por que os serviços devem sempre incluir uma interface de exceção usada para reportar defeitos e exceções aos clientes desses serviços.

31.5 Usando a mesma notação, estenda a Figura 31.6 para incluir definições para **MaxMinType** e **InDataFault**. As temperaturas devem ser representadas como números inteiros com um campo adicional que indique se a temperatura está em graus Fahrenheit ou em graus Celsius. **InDataFault** deve ser um tipo simples que consiste em um código de erro.

31.6 Defina uma especificação de interface para os serviços de avaliação de crédito do conversor de moeda e de cheque mostrados na Tabela 31.1.

31.7 Projete uma possível mensagem de entrada e de saída para os serviços mostrados na Figura 31.9. Você pode especificá-los em UML ou em XML.

31.8 Fornecendo razões para sua resposta, sugira dois importantes tipos de aplicações que provavelmente não façam uso de uma abordagem orientada a serviços.

31.9 Na Seção 31.2.1, apresentei um exemplo de uma empresa que desenvolveu um serviço de catálogo usado pelos sistemas de aquisição baseados na Web pelos clientes. Usando BPMN, projete um workflow que utilize esse serviço de catálogo para pesquisar e enviar pedidos para equipamentos computacionais.

31.10 Explique o que se entende por 'ação de compensação' e, usando um exemplo, mostre por que essas ações podem ser incluídas em workflows.

31.11 Para o exemplo do serviço de reserva de pacotes de viagem, projete um workflow que fará a reserva de transporte terrestre para um grupo de passageiros que chegam em um aeroporto. Eles devem ter a opção de reservar tanto um táxi quanto um carro alugado. Você pode supor que as empresas de táxi e de aluguel do carro oferecem serviços pela Web para fazer a reserva.

31.12 Usando um exemplo, explique em detalhes por que o teste detalhado de serviços que incluem ações de compensação é difícil.

32

Desenvolvimento de software orientado a aspectos

Objetivos

O objetivo deste capítulo é apresentar o desenvolvimento de software orientado a aspectos, que é baseado na idéia de separação de assuntos em módulos de sistema separados. Após ler este capítulo, você:

- compreenderá por que a separação de assuntos é um bom princípio-guia para o desenvolvimento de software;
- terá sido apresentado às idéias fundamentais em que se baseiam os aspectos e ao desenvolvimento de software orientado a aspectos;
- compreenderá como usar uma abordagem orientada a aspectos para engenharia de requisitos, projeto de software e programação;
- conhecerá os problemas de teste de sistemas orientados a aspectos.

Conteúdo

32.1 Separação de assuntos
32.2 Aspectos, pontos de junção e de corte
32.3 Engenharia de software com aspectos

Na maioria dos sistemas de grande porte, os relacionamentos entre os requisitos e componentes de programa são complexos. Um único requisito pode ser implementado por uma série de componentes e cada componente pode incluir elementos de vários requisitos. Isso significa que a implementação de uma mudança de requisito pode envolver o entendimento e mudança de muitos componentes. O reuso de componentes pode ser difícil devido a eles não implementarem uma única abstração de sistema, mas também incluírem fragmentos de código que implementam outros requisitos.

O desenvolvimento de software orientado a aspectos (AOSD — *Aspect-Oriented Software Development*), que apresentei brevemente no Capítulo 18, é uma abordagem emergente para desenvolvimento de software que tem a intenção de resolver esse problema e tornar os programas mais fáceis para serem mantidos e reusados. O AOSD é baseado em torno de um novo tipo de abstração chamado de *aspecto*. Aspectos são usados junto com outras abstrações como objetos e métodos. Eles englobam funcionalidades que atravessam e coexistem com outras funcionalidades

incluídas no sistema. Um programa executável orientado a aspectos é criado automaticamente pela composição de objetos, métodos e aspectos, de acordo com as especificações incluídas no código de programa-fonte.

O benefício principal de uma abordagem orientada a aspectos é que ela apóia a separação de assuntos. Conforme explico na Seção 32.1, a separação de assuntos em elementos independentes no lugar de incluir diferentes assuntos numa mesma abstração lógica é uma boa prática de engenharia de software. Pela representação de assuntos transversais como aspectos, esses assuntos podem ser compreendidos, reusados e modificados independentemente. Por exemplo, digamos que a autenticação de usuário é representada como um aspecto que solicita um nome de login e senha. Isso pode ser composto automaticamente em um programa sempre que a autenticação for necessária.

Uma importante característica de aspectos é que eles incluem uma definição de onde devem ser incluídos em um programa, bem como a implementação de código do assunto transversal. Assim, você pode especificar que o código transversal deve ser incluído antes ou depois de uma chamada de um método específico ou quando um atributo for acessado. Essencialmente, o aspecto é composto dentro do programa principal para criar um novo sistema maior.

Por exemplo, digamos que você tem um requisito que requeira uma autenticação de usuário antes que qualquer mudança em detalhes pessoais seja feita em um banco de dados. Você especifica isso em um aspecto definindo que o código de autenticação deve ser incluído antes de cada chamada dos métodos que atualizam detalhes pessoais. Subseqüentemente, você pode estender o requisito de autenticação de todas as atualizações de bancos de dados. Isso pode ser facilmente implementado pela modificação do aspecto por meio da mudança na definição de onde o código de autenticação deve ser composto no sistema. Você não precisa investigar ao longo do sistema procurando todas as ocorrências desses métodos. Portanto, é menos provável que você cometa erros e introduza acidentalmente vulnerabilidades de segurança no seu programa.

Pesquisas e desenvolvimentos em orientação a aspectos enfocaram preliminarmente a programação orientada a aspectos. Linguagens de programação orientada a aspectos como AspectJ (Kiczales, et al., 2001; Colyer e Clement, 2005; Laddad, 2003a; Laddad, 2003b; Colyer, et al., 2005) foram desenvolvidas e estenderam a programação orientada a objetos para incluir aspectos. Algumas empresas, como a IBM, estão iniciando o uso de programação orientada a aspectos nos processos de produção de software (Colyer e Clement, 2005). No entanto, foi reconhecido atualmente que assuntos transversais são igualmente problemáticos em outros estágios do processo de desenvolvimento de software. Pesquisadores estão investigando atualmente como utilizar a orientação a aspectos em engenharia de requisitos de sistema e projeto de sistema, e como testar e verificar programas orientados a aspectos.

O desenvolvimento de software orientado a aspectos não é ainda parte da engenharia de software principal. Não temos ainda princípios seguros e práticas testadas de AOSD, e estas levarão muitos anos de experiências práticas para virem à tona. Como toda tecnologia nova, ela prioriza o foco nos benefícios em vez de em problemas e custos. Enquanto acredito que o AOSD terá maior impacto nas práticas de engenharia de software, penso que isso acontecerá muitos anos antes de alguns problemas com essa abordagem serem resolvidos. No entanto, incluí uma explicação sobre o AOSD neste capítulo porque ela enfoca a separação de assuntos como um modo importante de pensar neles e sobre a estruturação de um sistema de software.

Nas seções seguintes do capítulo, portanto, explico alguns dos novos conceitos que são parte do AOSD e as vantagens e as desvantagens do uso de uma abordagem orientada a aspectos em diferentes estágios do processo de desenvolvimento de software. Como meu objetivo é ajudá-lo a compreender os conceitos em que se baseiam o AOSD, não entro em detalhes de nenhuma abordagem específica ou linguagem de programação orientada a aspectos.

32.1 Separação de assuntos

A separação de assuntos é um princípio principal do projeto e da implementação de software. Essencialmente, ele significa que você deve organizar o seu software de modo que cada elemento no programa (classe, método, procedimento etc.) faça uma coisa e somente uma coisa. Você pode então enfocar esse elemento sem considerar outros elementos do programa. Pode-se compreender cada parte do programa pelo conhecimento do seu assunto, sem necessitar entender outros elementos. Quando são necessárias mudanças, elas estão localizadas em um pequeno número de elementos.

A importância da separação de assuntos foi reconhecida nos estágios iniciais da história da ciência da computação. Sub-rotinas que englobam uma unidade de funcionalidade foram inventadas no início da década de 1950 e todos os programas subseqüentes que estruturavam mecanismos como procedimentos e classes de objetos foram projetados para fornecer melhores mecanismos para realizar a separação de assuntos. Contudo, todos esses mecanismos têm problemas em lidar com certos tipos de assuntos que atravessam outros assuntos. Os aspectos foram inventados para auxiliar a gerenciar os assuntos transversais.

Embora se concorde geralmente com que a separação de assuntos é uma boa prática de engenharia de software, é difícil definir qual é o significado real de um assunto. Algumas vezes, ele é definido como uma noção funcional na qual um assunto

é algum elemento funcional do sistema; alternativamente, ele pode ser definido de maneira mais abrangente como 'alguma parte de interesse ou foco de um programa'. Nenhuma dessas definições é particularmente útil. Os assuntos certamente são mais do que simples elementos funcionais e a definição mais geral é tão vaga que é praticamente inútil.

Na minha visão, muitas das tentativas em definir assuntos são problemáticas porque tentam relacionar assuntos a programas. Na verdade, conforme explicado por Jacobsen e Ng (2004), assuntos são realmente reflexões de requisitos do sistema e de prioridades dos stakeholders do sistema. O desempenho de sistema pode ser um assunto porque os usuários querem ter uma resposta rápida de um sistema; alguns stakeholders podem estar interessados em que o sistema inclua determinada funcionalidade; as empresas que estão dando suporte a um sistema podem estar interessadas em manutenção fácil. Um assunto é, portanto, algo de interesse ou significativo para um stakeholder ou para um grupo de stakeholder.

Com essa visão de um assunto, vemos então por que uma abordagem para implementação que separa assuntos em diferentes elementos de programa é uma boa prática. É mais fácil rastrear assuntos, definidos como requisito ou conjunto de requisitos relacionados e os componentes de programa que implementam esses assuntos. Se os requisitos mudam, a parte do programa que precisa ser mudada é óbvia.

Existem muitos tipos diferentes de assuntos de stakeholders:

1. Assuntos funcionais relacionados a uma funcionalidade específica a ser incluída em um sistema. Por exemplo, num sistema de controle de trens, um assunto funcional específico é o freio do trem.
2. Assuntos de qualidade de serviço que estão relacionados ao ambiente não funcional de um sistema. Esses assuntos incluem características como desempenho, confiabilidade e disponibilidade.
3. Assuntos de política relacionados às políticas gerais que governam o uso do sistema. Assuntos de política incluem assuntos de proteção e segurança e assuntos relacionados às regras de negócio.
4. Assuntos de sistema relacionados aos atributos do sistema como um todo, como sua facilidade de manutenção e de configuração.
5. Assuntos organizacionais relacionados aos objetivos e às prioridades organizacionais como a produção de um sistema dentro do orçamento, o uso de ativos de software existentes ou a manutenção de reputação da organização.

Os assuntos centrais de um sistema são esses assuntos funcionais que se relacionam a seu propósito primário. Portanto, para um sistema de informação de pacientes num hospital, os assuntos centrais são criação, edição, recuperação e gerenciamento de registros de pacientes. Em adição aos assuntos centrais, muitos dos sistemas de grande porte têm assuntos funcionais secundários. Esses assuntos podem envolver funcionalidades que compartilham informações com os assuntos centrais ou que sejam necessárias de modo que o sistema possa satisfazer seus requisitos não funcionais.

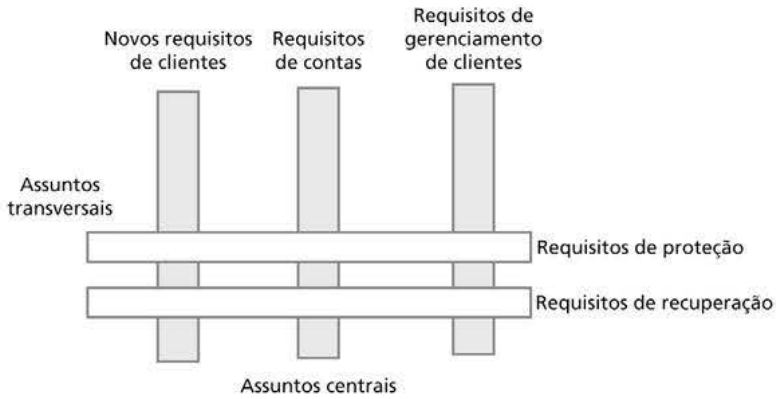
Como exemplo muito simples desse fato, considere um sistema que tenha um requisito para fornecer acesso concorrente a um buffer compartilhado. Explico um exemplo de um sistema como esse (Figura 15.14) no Capítulo 15, que aborda sistemas de tempo real. Esse buffer compartilhado é parte de um sistema de aquisição de dados no qual um processo produtor coloca dados no buffer e um processo consumidor extrai dados do buffer. Contudo, para garantir que os processos produtor e consumidor não interfiram um no outro, existe um assunto secundário essencial de sincronização. O sistema deve ser projetado de modo que o processo produtor não possa sobrescrever dados que não foram consumidos e o processo consumidor não possa extrair dados de um buffer vazio.

Em adição aos assuntos secundários, outros assuntos como de qualidades de serviço e assuntos de política refletem requisitos de sistema essenciais. Em geral, esses assuntos são de sistema — se aplicam ao sistema como um todo em vez de a requisitos individuais ou à realização dos requisitos em um programa. Chamamos esses assuntos de assuntos transversais para distingui-los de assuntos centrais. Assuntos funcionais secundários podem também ser transversais, embora nem sempre atravessem o sistema inteiro; antes, são associados com agrupamentos de assuntos centrais que fornecem funcionalidades relacionadas.

Isso é mostrado na Figura 32.1, baseada no exemplo de um sistema bancário de Internet. Esse sistema tem requisitos relacionados aos novos clientes como verificação de crédito e verificação de endereço. Ele também tem requisitos relacionados ao gerenciamento de clientes atuais e ao gerenciamento de contas de clientes. Todos esses requisitos são assuntos centrais quando são associados ao propósito primário do sistema — o fornecimento de um serviço bancário na Internet. No entanto, o sistema também tem requisitos de proteção baseados em políticas de proteção do banco e requisitos de recuperação para garantir que os dados não se percam no evento de falha de sistema. Os requisitos são assuntos transversais na medida em que podem influenciar a implementação de todos os outros requisitos de sistema.

Abstrações de linguagem de programação como procedimentos e classes são mecanismos que você normalmente usa para organizar a estrutura dos assuntos centrais de um sistema. Contudo, a implementação de assuntos centrais em linguagens de programação convencionais inclui geralmente código adicional para implementar os assuntos transversais funcionais, de qualidade de serviço e de política. Isso conduz a dois fenômenos indesejáveis: embaralhamento (*tagling*) e espalhamento (*scattering*).

Figura 32.1
Assuntos transversais.



O embaralhamento acontece quando um módulo de um sistema inclui código que implementa diferentes requisitos de sistema. O exemplo da Figura 32.2, que é parte do código para o sistema de buffer limitado mostrado na Figura 15.14, ilustra esse fenômeno. O código que apóia o assunto primário (nesse caso, colocar um registro no buffer) é embaralhado com o código que implementa a sincronização. O código de sincronização que está associado com o assunto secundário para garantir a exclusão mútua tem de ser incluído em todos os métodos que acessam o buffer compartilhado. O código associado com o assunto de sincronização é mostrado como código sombreado na Figura 32.2.

O fenômeno relacionado ao espalhamento ocorre quando a implementação de um único assunto (um requisito lógico ou conjunto de requisitos) é espalhada em diversos componentes de um programa. É provável que isso ocorra quando requisitos relacionados aos assuntos secundários ou assuntos de política são implementados.

Por exemplo, digamos que um sistema de gerenciamento de registros de hospital tem uma série de componentes dedicados à gerência de informações de pessoal, medicamentos, consultas, imagens médicas, diagnósticos e tratamentos. Eles podem ser configurados de formas diferentes por tipos diferentes de clínicas e implementam os assuntos centrais do sistema — a manutenção de registros de pacientes.

Contudo, considere que haja também um assunto secundário importante, que é a manutenção de informações estatísticas — o hospital deseja registrar detalhes de como diversos pacientes foram admitidos e liberados a cada mês, como muitos pacientes morreram, quais medicamentos foram administrados, as razões para as consultas etc. Esses requisitos têm de ser implementados por adição de código que torna os dados anônimos (para manter a privacidade de pacientes) e os escreve em um banco de dados estatísticos. Os componentes estatísticos processam os dados estatísticos e geram relatórios estatísticos necessários.

Isso está ilustrado na Figura 32.3. Esse diagrama mostra exemplos de três classes que poderiam ser incluídas no sistema de registros de pacientes com alguns métodos centrais para a gerência de informações de pacientes. A área sombreada mostra os métodos necessários para implementar um assunto estatístico secundário. Você pode ver que esse assunto estatístico é espalhado por meio de outros assuntos centrais.

Os problemas com o espalhamento e embaralhamento surgem quando os requisitos iniciais do sistema mudam. Por exemplo, digamos que novos dados estatísticos tiveram de ser coletados no sistema de registros de pacientes. As mudanças no sistema não estão todas localizadas num único lugar e, assim, você precisa despender tempo para procurar todos os componentes do sistema que têm de ser alterados. Você precisa então alterar cada um desses componentes para incorporar as mudanças necessárias. Devido ao tempo despendido, é um processo oneroso. Há sempre a possibilidade de que você

Figura 32.2
Embaralhamento de gerenciamento de buffer e código de sincronização.

```
synchronized void put (SensorRecord rec) throws InterruptedException
{
    if (numberOfEntries == bufsize)
        wait () ;
    store [back] = new SensorRecord (rec.sensorId, rec.sensorVal) ;
    back = back + 1 ;
    if (back == bufsize)
        back = 0 ;
    numberOfEntries = numberOfEntries + 1 ;
    notify () ;
} // put
```

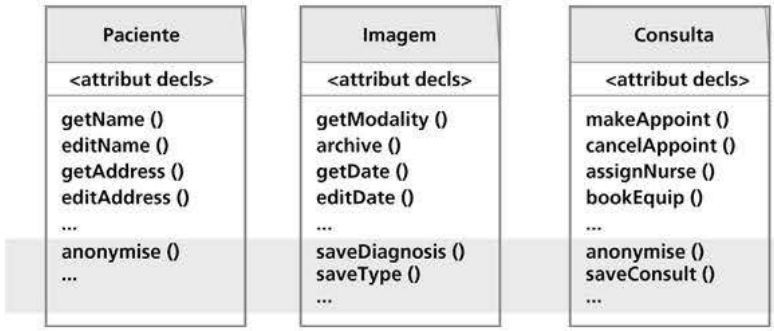


Figura 32.3 Espalhamento de métodos que implementam assuntos secundários.

esquecerá algum código que deve ser alterado e, portanto, a estatística será incorreta. Além disso, quando muitas mudanças têm de ser feitas, isso aumenta as chances de que você cometerá um erro ou introduzirá defeitos no software.

32.2 Aspectos, pontos de junção e de corte

Nesta seção, apresento os novos conceitos mais importantes associados com o desenvolvimento de software orientado a aspectos e ilustro esses conceitos com exemplos de um sistema de informação de pacientes de hospital. A terminologia que uso foi primeiro introduzida pelos desenvolvedores do AspectJ no fim da década de 1990. No entanto, os conceitos são geralmente aplicáveis e não específicos para essa linguagem de programação. A Tabela 32.1 sumariza os termos principais que você precisa compreender.

O sistema de informação de pacientes de hospital, introduzido na Seção 32.1, inclui um número de componentes diferentes que trata informações de pacientes logicamente relacionadas. O componente paciente mantém informações pessoais sobre um paciente, o componente medicação mantém informações sobre medicamentos que podem ser prescritos etc. Ao projetar o sistema usando uma abordagem baseada em componentes, instâncias diferentes do sistema podem ser configuradas. Por exemplo, uma versão poderia ser configurada para cada tipo de clínica com médicos autorizados para prescrever somente medicações relevantes para essa clínica. Isso simplifica o trabalho do pessoal da clínica e reduz as chances de que o médico prescreverá a medicação errada.

Contudo, essa organização significa que as informações são atualizadas no banco de dados com base em uma série de locais diferentes no sistema. Por exemplo, as informações de pacientes podem ser alteradas quando houver modificação de detalhes pessoais, quando sua medicação for alterada, quando eles forem designados para um novo especialista etc. Para simplificar, suponha que todos os componentes do sistema usam uma estratégia de denominação consistente e que todas as atualizações na base de dados são implementadas pelos métodos que iniciam com 'update'. Existem, portanto, métodos no sistema como:

updatePersonalInformation (patientId, infoupdate)
updateMedication (patientId, medicationupdate)

Tabela 32.1 Terminologia usada em desenvolvimento de software orientado a aspectos

Termo	Definição
Recomendação	Código que implementa um aspecto.
Aspecto	Abstração de programa que define um assunto transversal. Inclui a definição de um ponto de corte e a recomendação associada a esse assunto.
Ponto de junção	Evento em um programa em execução no qual a recomendação associada a um aspecto pode ser executada.
Modelo de ponto de junção	Conjunto de eventos que pode ser referenciado em um ponto de corte.
Ponto de corte	Declaração, incluída em um aspecto, que define os pontos de junção em que a recomendação de aspecto associada deve ser executada.
Composição	Incorporação de código de recomendação nos pontos de junção especificados por um compositor de aspectos.

O paciente é identificado pelo **patientId** e as mudanças a serem feitas são codificadas no segundo parâmetro — os detalhes não são importantes para esse exemplo. As atualizações são feitas pelo pessoal da clínica ligado no sistema.

Imagine que ocorre uma brecha de proteção quando as informações de pacientes são maliciosamente alteradas. Talvez alguém tenha acidentalmente deixado o seu computador ligado e uma pessoa não autorizada tenha ganhado acesso ao sistema. Alternativamente, um intruso autorizado possa ter obtido acesso e maliciosamente alterado as informações de pacientes. Para reduzir a probabilidade de que isso aconteça novamente, uma nova política de proteção é introduzida. Antes de qualquer alteração ser feita no banco de dados de pacientes, a pessoa que solicitou a mudança deve se autenticar novamente no sistema. Detalhes de quem fez a alteração são também registrados em um log em um arquivo separado. Isso auxilia rastrear problemas se eles ocorrerem novamente.

Uma maneira de implementar essa nova política é modificar o método **update** em cada componente para chamar outros métodos que fazem a autenticação e o registro. Alternativamente, o sistema poderia ser modificado de modo que, cada vez que um método de atualização fosse chamado, chamadas de métodos são adicionados antes da chamada do método de atualização para fazer a autenticação e depois para registrar as mudanças feitas no log. Nenhuma dessas soluções é muito boa para esse problema:

1. A primeira abordagem conduz a uma implementação embaralhada. Logicamente, a atualização de banco de dados, a autenticação de quem originou uma atualização e o registro de detalhes de atualização no log são assuntos separados e não relacionados. Você pode desejar incluir a autenticação em qualquer lugar no sistema sem log ou com log de ações separado da ação de atualização. O mesmo código de autenticação e de registro precisa ser incluído em vários métodos diferentes.
2. A abordagem alternativa conduz a uma implementação espalhada. Se você inclui explicitamente as chamadas de método para fazer a autenticação e o registro antes e depois de toda chamada de métodos de atualização, esse código está incluído em muitos lugares diferentes no sistema.

A autenticação e o registro atravessam os assuntos centrais do sistema e podem ser incluídos em muitos locais diferentes. Em um sistema orientado a aspectos, você pode representar os assuntos transversais como aspectos separados. Um aspecto inclui uma especificação de onde o assunto transversal será composto no programa e o código para implementar o assunto. Isso está ilustrado na Figura 32.4, que define um aspecto de autenticação. A notação que uso nesse exemplo segue o estilo do AspectJ, mas usa uma sintaxe simplificada, que deve ser compreensível sem o conhecimento de Java ou de AspectJ.

Aspectos são completamente diferentes de outras abstrações de programas, sendo que os aspectos em si incluem uma especificação de onde devem ser executados. Em outras abstrações, como métodos, há uma separação clara entre a definição de uma abstração e o seu uso. Você pode chamar um método com base em qualquer lugar no escopo e não é possível saber de onde o método deve ser chamado apenas olhando para ele. Aspectos, de outro modo, incluem um *ponto de corte* — uma declaração que define onde o aspecto será composto no programa.

Figura 32.4

Descrição de um aspecto de autenticação.

```
aspect autenticação
{
    before: call (public void update* (..)) // isto é um ponto de corte
    {
        // isto é a recomendação que deve ser executada quando composto no
        // sistema em execução
        int tries = 0 ;
        string userPassword = Password.Get (tries) ;
        while (tries < 3 && userPassword != thisUser.password ( ) )
        {
            // allow 3 tries to get the password right
            tries = tries + 1 ;
            userPassword = Password.Get (tries) ;
        }
        if (userPassword != thisUser.password ( )) then
            //se a senha estiver errada, supor que o usuário esqueceu-se de executar logout
            System.Logout (thisUser.uid) ;
    }
} // autenticação
```

Neste exemplo, o ponto de corte é uma declaração simples:

```
before: call (public void update* (..))
```

O significado disso é que antes da execução de qualquer método cujo nome inicia com a string **update** seguido por qualquer outra string (o caractere * é chamado de *wildcard* e combina qualquer string de caracteres seguidos em identificadores), o código depois da definição do ponto de corte deve ser executado. Esse código é chamado de recomendação e é a implementação do assunto transversal. Nesse caso, a recomendação obtém uma senha da pessoa que solicita a mudança e verifica se ela coincide com a senha do usuário que está ligado no momento. Se não, o usuário é desligado e a atualização não é processada.

A habilidade para especificar, usando pontos de corte, onde o código deve ser executado é a característica diferente dos aspectos. Contudo, para compreender o que significa pontos de corte, você necessita compreender outro conceito — a idéia de um *ponto de junção*. Um ponto de junção é um evento que ocorre durante a execução de um programa — assim, ele poderia ser uma chamada de método, a iniciação de uma variável, a atualização de um campo etc. Obviamente, há muitos tipos diferentes de eventos que ocorrem durante a execução de programas. Um *modelo de ponto de junção* define esses eventos que podem ser referenciados pelo programador orientado a aspectos. Por exemplo, em AspectJ, os eventos que são parte de um modelo de ponto de junção incluem:

- eventos de chamada — chama um método ou um construtor;
- eventos de execução — execução de um método ou um construtor;
- eventos de iniciação — iniciação de classe ou objeto;
- eventos de dados — acesso ou atualização de um campo;
- eventos de exceção — tratamento de uma exceção.

Um ponto de corte identifica eventos específicos (por exemplo, a chamada de um procedimento identificado) com o qual a recomendação deve ser associada. Isso significa que você pode compor a recomendação no programa em diferentes contextos. Por exemplo:

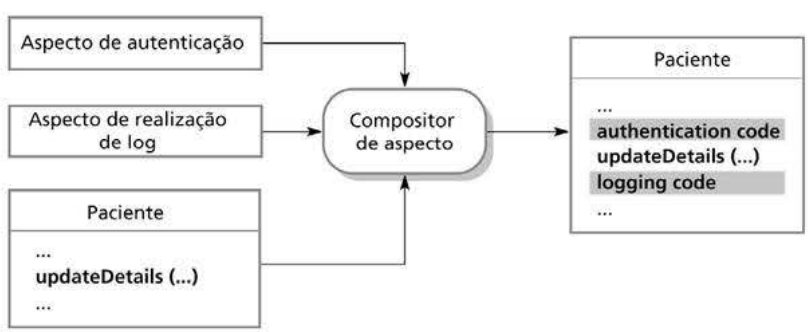
1. Antes da execução de um método específico, uma lista de métodos identificados ou uma lista de métodos cujos nomes coincidam com uma especificação de padrão (como update*).
2. Depois do retorno normal ou excepcional de um método. No exemplo mostrado na Figura 32.4, você poderia definir um ponto de corte que executaria o código de registro depois de todas as chamadas de métodos de atualização.
3. Quando um campo de um objeto é modificado, você pode incluir uma recomendação para monitorar a alteração desse campo.

A inclusão da recomendação nos pontos de junção especificados nos pontos de corte é de responsabilidade de um compositor de aspecto. Os compositores de aspectos são extensões para os compiladores que processam a definição de aspectos e as classes de objetos e métodos que definem o sistema. O compositor gera então um novo programa com os aspectos incluídos nos pontos de junção especificados. Os aspectos são integrados de modo que os assuntos transversais são executados nos locais certos no sistema final.

A Figura 32.5 ilustra essa composição de aspectos para os aspectos de autenticação e de registro que devem ser incluídos no sistema de gerenciamento de pacientes. Há três abordagens diferentes para a composição de aspectos:

1. Pré-processamento de código-fonte, no qual o compositor obtém o código-fonte e gera um novo código em uma linguagem do tipo Java ou C++, que pode então ser compilado por meio de um compilador de linguagem padrão. Essa abordagem foi adotada para a linguagem AspectX com seu XWeaver associado (Birrer, et al., 2005).
2. Composição em tempo de ligação na qual o compilador é modificado para incluir um compositor de aspectos. Uma linguagem orientada a aspectos como AspectJ é processada e um bytecode Java padrão é gerado. Pode ser então

Figura 32.5
Composição de aspectos.



executada diretamente por um interpretador Java ou processada futuramente para gerar um código de máquina nativo. No momento da escrita deste livro, essa era a abordagem mais amplamente usada.

3. Composição dinâmica no tempo de execução. Essa é a abordagem mais geral, mas também com o maior custo indireto. Nesse caso, os pontos de junção são monitorados e quando um evento que está referenciado em um ponto de corte ocorre, a recomendação correspondente é integrada ao programa em execução.

32.3 Engenharia de software com aspectos

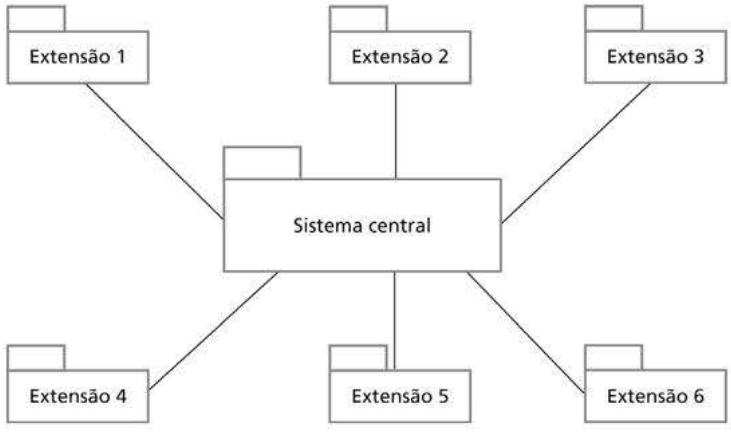
Os aspectos foram inicialmente introduzidos como uma construção de linguagem de programação, mas, conforme expliquei, o conceito de assuntos é um daqueles que vem de requisitos de sistema. Portanto, faz sentido adotar uma abordagem orientada a aspectos em todos os estágios do processo de desenvolvimento de sistema. Nos estágios iniciais de engenharia de software, a adoção de uma abordagem orientada a aspectos significa usar o conceito de separação de assuntos como base para pensar em requisitos e o projeto de sistemas. A identificação e a modelagem de assuntos devem ser parte dos processos de engenharia de requisitos e projeto. As linguagens de programação orientadas a aspectos fornecem, portanto, o apoio tecnológico para manter a separação de assuntos na sua implementação do sistema.

Jacobsen e Ng (2004) sugeriram que você deve pensar num sistema que apóie diferentes assuntos envolvidos como um sistema central mais extensões. Ilustrei isso na Figura 32.6, na qual usei packages UML para representar tanto o centro quanto as extensões. O sistema central é o conjunto de características de sistema que fornecem apoio para o propósito essencial do sistema. Portanto, se o propósito do sistema é manter a informação de pacientes em um hospital, o sistema fornece um meio para criar, editar, gerenciar e acessar um banco de dados de registros de pacientes. As extensões para o sistema central refletem assuntos adicionais dos stakeholders do sistema que devem ser integrados com o sistema central. Por exemplo, é importante que o sistema de informação de hospital mantenha a confidencialidade de informações de pacientes e, assim, uma extensão possa ser dedicada ao controle de acesso, outra à criptografia etc.

Há uma quantidade de tipos de extensões derivadas de diferentes tipos de assuntos, conforme expliquei na Seção 32.1.

1. *Extensões funcionais secundárias*. Essas extensões adicionam capacidades funcionais para funcionalidades fornecidas no sistema central. Por exemplo, a produção de relatórios de medicamentos prescritos no mês anterior seria uma extensão funcional secundária para um sistema de informação de pacientes.
2. *Extensões de políticas*. Essas extensões adicionam capacidades funcionais para apoiar algumas políticas organizacionais. Extensões que adicionam características de proteção são exemplos de extensões de políticas.
3. *Extensões de QoS*. Essas extensões adicionam capacidades funcionais para auxiliar a atender os requisitos da qualidade de serviço (QoS — *Quality of Service*) que foram especificados para o sistema. Por exemplo, uma extensão poderia fornecer apoio a um cache para reduzir o número de acessos ao banco de dados ou a back-ups automáticos para se recuperar no evento de uma falha de sistema.
4. *Extensões de infra-estrutura*. Essas extensões adicionam capacidades funcionais para dar suporte à implementação de um sistema em alguma plataforma específica. Por exemplo, em um sistema de informação de pacientes, as extensões de infra-estrutura poderiam ser usadas para implementar a interface para o sistema de gerenciamento de banco de dados. Se isso mudar, as mudanças poderão ser feitas por meio da alteração de extensões de infra-estrutura associadas.

Figura 32.6
Sistema central com extensões.



As extensões adicionam sempre algum tipo de funcionalidade ou características para o sistema central. Aspectos são um meio para implementar as extensões e podem ser compostos com a funcionalidade do sistema central usando os recursos de composição no ambiente de programação orientado a aspectos.

32.3.1 Engenharia de requisitos orientada a assuntos

Conforme sugeri na Seção 32.1, os assuntos refletem os requisitos dos stakeholders. Tais assuntos podem refletir a funcionalidade exigida por um stakeholder, a qualidade de serviço de sistema, políticas organizacionais ou questões que estejam relacionadas aos atributos do sistema como um todo. Portanto, faz sentido adotar uma abordagem para a engenharia de requisitos que identifica e especifica os diferentes assuntos dos stakeholders. O termo *aspecto inicial* é algumas vezes usado para fazer referência ao uso de aspectos nos estágios iniciais do ciclo de vida do software, nos quais a separação de assuntos é enfatizada.

A importância da separação de assuntos durante a engenharia de requisitos foi reconhecida durante muitos anos. Os pontos de vista, que abordei na Seção 7.2.1 e que foram incorporados em uma série de métodos de engenharia de requisitos (Finkelstein, et al., 1992; Easterbrook e Nuseibeh, 1996; Kotonya e Sommerville, 1996), são uma maneira de separar os assuntos de diferentes stakeholders. Os assuntos refletem a funcionalidade distinta exigida por diferentes grupos de stakeholders. Contudo, existem também requisitos que atravessam todos os pontos de vista conforme mostrado na Figura 32.7. O diagrama mostra que os pontos de vista podem ser de tipos diferentes, mas os assuntos transversais (como regulamentações, confiabilidade e proteção) geram requisitos que podem impactar todos os pontos de vista de sistema. Essa foi a grande consideração no trabalho que fiz no desenvolvimento do método PreView (Sommerville e Sawyer, 1997; Sommerville, et al., 1998), que incluiu passos para identificar os assuntos transversais e não funcionais.

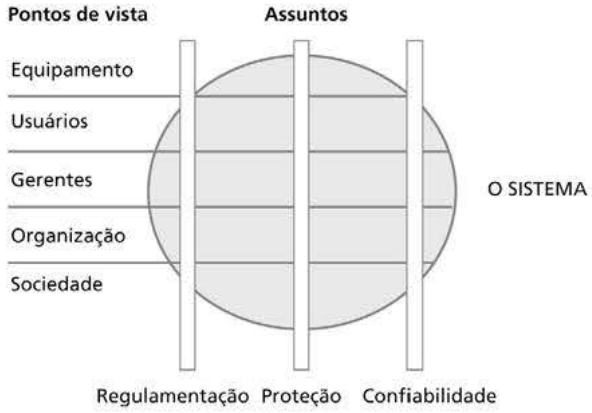
Para desenvolver um sistema organizado como mostrado na Figura 32.6, você deve identificar os requisitos do sistema central mais os requisitos para as extensões do sistema. Uma abordagem orientada a pontos de vista para a engenharia de requisitos, na qual cada ponto de vista represente os requisitos relacionados a grupos de stakeholders, é uma maneira de separar assuntos centrais e secundários. Se você organizar os requisitos de acordo com o ponto de vista do stakeholder, pode analisá-los para descobrir os requisitos relacionados que aparecem em todos ou na maioria dos pontos de vista. Esses requisitos representam a funcionalidade central do sistema. Outros requisitos podem ser requisitos de assuntos separados que podem ser implementados como extensões da funcionalidade central.

Por exemplo, o Quadro 32.1 mostra requisitos delineados para três possíveis pontos de vista de um sistema usado para manter o controle de equipamento especializado utilizado em serviços de emergência. Esse equipamento está localizado em diferentes lugares da região ou do estado e, numa situação de emergência como uma inundação ou um terremoto, os serviços de emergência usam o sistema para descobrir qual equipamento está disponível num lugar mais próximo do problema.

Você pode perceber com base neste exemplo que os stakeholders de todos os pontos de vista distintos precisam ser capazes de encontrar equipamentos específicos, procurar os equipamentos disponíveis em cada localização, realizar check-in/check-out de equipamentos no depósito. Esses são, portanto, requisitos do sistema central. Os requisitos secundários auxiliam no apoio às necessidades mais específicas em cada ponto de vista. Existem requisitos secundários para as extensões do sistema que apóiam o uso, gerenciamento e manutenção de equipamentos.

Os requisitos funcionais secundários identificados com o uso de pontos de vista não necessariamente atravessam os requisitos de todos os outros pontos de vista. Eles são mais focados em fornecer uma separação de assuntos. Além deles, há outros assuntos transversais que geram requisitos importantes para todos os pontos de vista. Esses requisitos refletem freqüentemente os requisitos de política e de qualidade de serviço que se aplicam ao sistema como um todo. Os assuntos

Figura 32.7
Pontos de vista e assuntos.



Quadro 32.1 Pontos de vista sobre um sistema de inventário de equipamentos.

1. **Usuários de serviços de emergência**
   - 1.1 Encontrar um tipo de equipamento especificado (por exemplo, engrenagem de levantamento de peso)
   - 1.2 Visualizar equipamento disponível em um depósito especificado
   - 1.3 Check-out de equipamento
   - 1.4 Check-in de equipamento
   - 1.5 Acomodar equipamento para ser transportado para emergência
   - 1.6 Enviar relatório de danos
   - 1.7 Encontrar depósito próximo à emergência
2. **Planejadores de emergência**
   - 2.1 Encontrar um tipo de equipamento especificado
   - 2.2 Visualizar equipamento disponível em uma localização especificada
   - 2.3 Adicionar e remover equipamento de um depósito
   - 2.4 Mover equipamento de um depósito para o outro
   - 2.5 Pedir novo equipamento
3. **Pessoal de manutenção**
   - 3.1 Check-in/check-out de equipamento para manutenção
   - 3.2 Visualizar equipamento disponível em cada depósito
   - 3.3 Encontrar um tipo de equipamento especificado
   - 3.4 Visualizar cronograma de manutenção de um equipamento
   - 3.5 Preencher registro de manutenção de um equipamento
   - 3.6 Mostrar todos os itens em um depósito que precisam de manutenção

transversais podem emergir de requisitos de pontos de vista individuais relevantes a todos os pontos de vista ou de discussões gerais de propriedades globais de sistema.

No sistema de inventário de equipamentos, um exemplo de um assunto transversal é a necessidade de disponibilidade de sistema. Emergências podem acontecer com pouco ou nenhum aviso. Salvar vidas pode exigir que equipamentos essenciais estejam prontos tão rápido quanto possível. Deve haver, portanto, requisitos do sistema de inventário de equipamentos com alto nível de disponibilidade de sistema. Alguns exemplos desses requisitos, com a razão associada, são mostrados no Quadro 32.2. Com base nesses requisitos, você pode então identificar extensões necessárias da funcionalidade central para o registro de transações e relatório de status. Estes tornam mais fáceis a identificação de problemas e o chaveamento para um sistema de back-up.

O resultado do processo de engenharia de requisitos deve ser um conjunto de requisitos estruturados em torno da noção de um sistema central mais extensões. Por exemplo, no sistema de inventário, exemplos de requisitos centrais poderiam ser:

C.1 O sistema permitirá aos usuários autorizados visualizar a descrição de qualquer item de equipamento no inventário de serviços de emergência.

C.2 O sistema incluirá um recurso de busca para permitir aos usuários autorizados buscar tanto inventários individuais quanto um inventário completo para um item específico de equipamento ou um tipo específico de equipamento.

O sistema pode também incluir explicitamente uma extensão para dar suporte a aquisição e substituição de equipamentos. Portanto, os requisitos para essa extensão poderiam ser:

E1.1 Será possível para usuários autorizados colocar pedidos com fornecedores credenciados para itens de reposição de equipamento.

Quadro 32.2

Requisitos relacionados à disponibilidade do sistema de inventário de equipamentos.

AV.1	Haverá um sistema 'hot standby' disponível em uma localização geograficamente bastante separada do sistema principal.
Razão:	A emergência pode afetar a localização principal do sistema.
AV.1.1	Todas as transações serão registradas no local do sistema principal e no local do sistema standby remoto
Razão:	Isso permite que as transações sejam reproduzidas e o banco de dados do sistema esteja consistente
AV.1.2	O sistema enviará informações de status para o sistema da sala de controle de emergência a cada cinco minutos.
Razão:	Os operadores do sistema da sala de controle podem alternar para 'hot standby' caso o sistema principal esteja indisponível.

E1.1.1 Quando um item de equipamento for pedido, ele deve ser alocado em um inventário específico e marcado nesse inventário como 'sob solicitação'.

Nesse estágio, é importante que você não tenha muitos assuntos ou extensões para o sistema. Eles só confundem o leitor e podem conduzir a um projeto prematuro. Isso limita a liberdade dos projetistas e pode resultar em um projeto de sistema que pode não atender a seus requisitos de qualidade de serviço.

32.3.2 Projeto e programação orientados a aspectos

O projeto orientado a aspecto é o processo de projeto de um sistema que faz uso dos aspectos para implementar os assuntos transversais e as extensões identificados durante o processo de engenharia de requisitos. Nesse estágio, você precisa traduzir os assuntos que se relacionam aos problemas a serem resolvidos em aspectos correspondentes no programa que implementa a solução. Você também precisa compreender como esses aspectos serão compostos com outros componentes do sistema e assegurar que não surjam ambigüidades de composição.

Uma declaração de alto nível de requisitos provê uma base para identificação de algumas extensões de sistema que podem ser implementadas como aspectos. Você precisa então desenvolver mais detalhes para identificar outras extensões e compreender a funcionalidade necessária. Uma maneira de fazer isso é identificar um conjunto de casos de uso (explicado no Capítulo 7) associado com cada ponto de vista. Os modelos de caso de uso são a ponte entre os requisitos e o projeto. São mais detalhados do que os requisitos de usuário. Em um modelo de caso de uso, você descreve os passos de cada interação de usuário e, assim, inicia a identificação e a definição de classes no sistema.

Jacobsen e Ng (2004) escreveram um livro que explica como os casos de uso podem desempenhar um papel fundamental na engenharia de software orientada a aspectos. Eles apresentam uma explicação abrangente (a qual não tenho espaço para abordar em detalhes) de como a abordagem de caso de uso pode servir como base para a engenharia de software orientada a aspectos. Sugerem que cada caso de uso represente um aspecto e propõem extensões para a abordagem de caso de uso para dar suporte a pontos de junção e pontos de corte. Introduzem também a noção de fatias de caso de uso e módulos de caso de uso que incluam fragmentos de classes que implementam um aspecto, que podem ser compostos para criar um sistema completo.

A Figura 32.8 mostra exemplos de três casos de uso que poderiam ser parte de um sistema de gerenciamento de inventário. Eles refletem os assuntos de adição de equipamento a um inventário e pedido de equipamento. O pedido de equipamento e a adição de equipamento ao depósito são assuntos relacionados. Uma vez que os itens pedidos foram entregues, devem ser adicionados ao inventário e entregues a um dos depósitos de equipamentos.

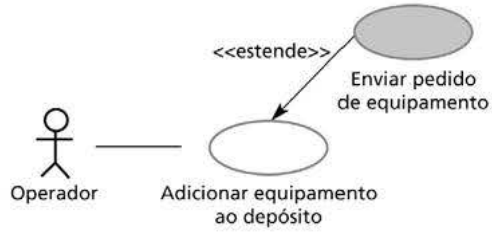
A UML já inclui a noção de extensão de casos de uso na qual estende a funcionalidade para outro caso de uso. A Figura 32.9 mostra como a colocação de um pedido de equipamento estende o caso de uso central para a adição de equipamento em um depósito específico. Se o equipamento a ser adicionado não existe, ele pode ser pedido e adicionado

Figura 32.8

Casos de uso do sistema de gerenciamento de inventário.

Figura 32.9

Casos de uso de extensão.



no depósito quando o equipamento for entregue. Durante o desenvolvimento dos modelos de caso de uso, você deve procurar características comuns nos modelos de caso de uso e, quando for possível, estruturar os casos de uso como caso de uso central mais extensões. As características transversais, como o registro de todas as transações, podem ser também representadas como extensões de casos de uso. Jacobsen e Ng explicam como extensões desse tipo podem ser implementadas como aspectos.

O desenvolvimento de um processo efetivo para projeto orientado a aspectos é essencial se o projeto orientado a aspectos deve ser aceito e usado. Sugiro que um processo de projeto orientado a aspectos deva incluir as atividades mostradas na Figura 32.10. Essas atividades são:

1. *Projeto de sistema central*. Nesse estágio, você deve projetar uma arquitetura para dar suporte à funcionalidade central, núcleo do sistema. A arquitetura deve também levar em conta os requisitos de qualidade de serviço como os requisitos de desempenho e de confiança. No entanto, o suporte funcional específico necessário deve ser implementado como extensões.
2. *Identificação e projetos de aspectos*. Iniciando com as extensões identificadas nos requisitos de sistema, você deve analisá-los para ver se são aspectos em si mesmos ou se devem ser quebrados em diversos aspectos. Uma vez que os aspectos foram identificados, podem então ser projetados separadamente, levando em conta o projeto das características do sistema central.
3. *Projeto de composição*. Nesse estágio, você analisa o sistema central e os projetos de aspectos para descobrir onde os aspectos devem ser compostos com o sistema central. Essencialmente, nesse estágio, você está identificando os pontos de junção no programa onde os aspectos serão compostos.
4. *Análise e resolução de conflitos*. Um problema com aspectos é que eles podem interferir um com outro quando são compostos com o sistema central. Os conflitos surgem quando um ponto de corte conflita com aspectos diferentes que especificam que eles devem ser compostos no mesmo ponto do programa. Contudo, outros conflitos mais sutis podem também surgir. Quando os aspectos são projetados independentemente, é possível formular hipóteses sobre a funcionalidade do sistema central que deve ser modificada. Contudo, quando muitos aspectos são compostos, a funcionalidade do sistema pode ser afetada de modo que as hipóteses iniciais tornam-se inválidas.
5. *Projeto de nomes*. Importante atividade de projeto que define padrões para denominação de entidades no programa. É essencial para evitar o problema de pontos de corte acidentais. Ocorre quando, em algum ponto de junção de programa, o nome coincide com algum padrão de ponto de corte, mas não há intenção de compor uma recomendação nesse momento. Uma vez que você projetou os padrões de nomes, pode ter de fazer modificações no seu modelo de projeto para renomear os elementos de projeto.

Esse processo é, naturalmente, um processo iterativo no qual você faz propostas iniciais de projeto e então refina as propostas ao analisar e compreender as questões de projeto. Normalmente, você espera refinar as extensões identificadas nos requisitos para um maior número de aspectos.

O resultado de um processo orientado a aspectos é um modelo de projeto orientado a aspectos. Pode ser expresso numa versão estendida de UML, a qual inclui novas construções específicas de aspecto, como as propostas por Clarke e Baniassad (2005) e Jacobsen e Ng (2004). Os elementos essenciais de 'aspecto UML' são alguns meios de modelagem de aspectos e um meio de especificação de pontos de junção nos quais a recomendação de aspecto deve ser composta com o sistema central. No entanto, nenhum dos padrões para inclusão de aspectos na UML ainda foi acordado.

A Figura 32.11 é um exemplo de modelo de projeto orientado a aspectos. Eu usei o estereótipo da UML para um aspecto proposto por Jacobsen e Ng. A Figura 32.11 mostra o sistema central para inventário de serviços de emergência mais alguns aspectos que poderiam ser compostos com esse núcleo. Mostrei algumas classes do sistema central e alguns aspectos. Este

Figura 32.10 Processo genérico de projeto orientado a aspectos.

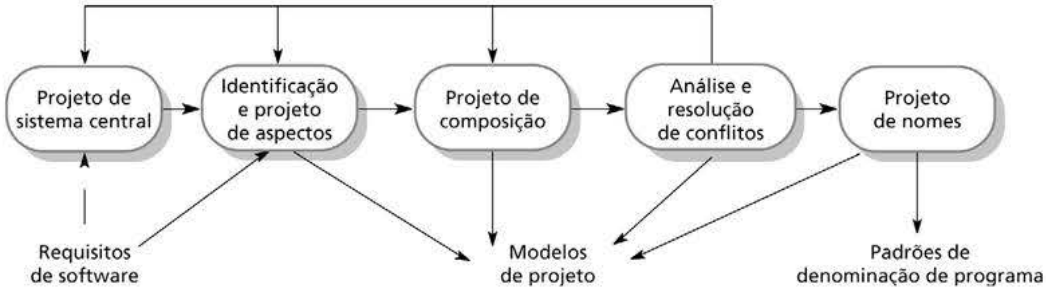
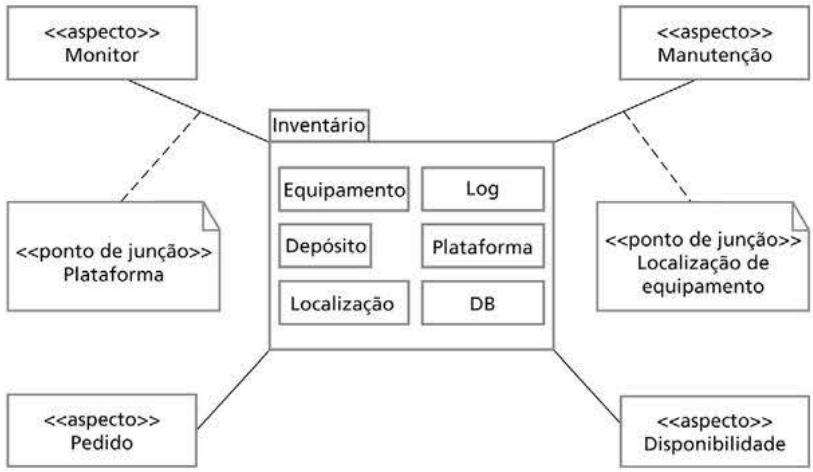


Figura 32.11
Modelo de projeto orientado a aspectos.



é um quadro simplificado — um modelo completo incluiria mais classes e aspectos. Perceba como usei as notas da UML para prover informações adicionais sobre as classes transversais a alguns aspectos.

A Figura 32.12 é uma versão expandida de um modelo de aspecto que revela muita informação sobre o aspecto. Obviamente, antes de você projetar aspectos, deve ter o projeto do sistema central. Como eu não tenho espaço para mostrar isso aqui, elaborei diversas hipóteses sobre classes e métodos no sistema central.

A primeira seção do aspecto estabelece os pontos de corte que especificam onde ele será composto com o sistema central. Por exemplo, o primeiro ponto de corte especifica que o aspecto pode ser composto no ponto de junção **call getItemInfo (..)**. A seção seguinte define as extensões implementadas pelo aspecto. Nesse exemplo, a declaração de extensão pode ser lida como:

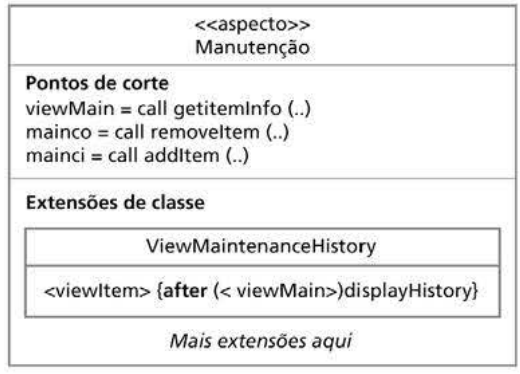
No método **viewItem**, *depois da chamada do método* **getItemInfo**, *uma chamada para o método* **displayHistory** *deve ser incluída para apresentar o registro de manutenção*

A programação orientada a aspectos iniciou-se nos laboratórios PARC da Xerox, em 1997, com a invenção da linguagem de programação AspectJ. Esta permanece sendo a linguagem orientada a aspectos mais amplamente usada, mas as extensões orientadas a aspectos de outras linguagens, como C# e C++, foram implementadas. Outras linguagens experimentais foram também desenvolvidas para dar suporte à separação explícita de assuntos e a sua composição. Não entrarei em detalhes de programação orientada a aspectos aqui porque isso é extensivamente tratado em outros livros (Laddad, 2003b; Gradecki e Lezeiki, 2003; Colyer et al., 2005).

Se você seguiu a abordagem orientada a aspectos para projetar o seu sistema, já terá identificado a funcionalidade central e as suas extensões a serem implementadas como aspectos transversais. O foco do processo de programação deve ser então escrever código que implementa a funcionalidade central e a sua extensão e, criticamente, especificar os pontos de corte nos aspectos de modo que a recomendação de aspecto seja composta no código-base nos lugares corretos.

A especificação correta de pontos de corte é muito importante na medida em que eles definem onde a recomendação de aspecto será composta com a funcionalidade central. Se você comete um erro na especificação de ponto de corte, a recomendação de aspecto será composta no local errado do programa. Isso pode conduzir a um comportamento inesperado e imprevisível do programa. A aderência aos padrões de denominação estabelecidos durante o projeto do sistema é essen-

Figura 32.12
Modelo parcial de um aspecto.



cial. Você tem também de revisar todos os aspectos para assegurar que a interferência entre aspectos não ocorrerá se dois ou mais aspectos forem compostos no sistema central no mesmo ponto de junção. De maneira geral, é melhor evitar isso completamente, mas, ocasionalmente, ela pode ser a coisa certa fazer. Nessas circunstâncias, você deve assegurar que os aspectos são completamente independentes. O comportamento do programa não deve depender da ordem de composição dos aspectos no programa.

32.3.3 Verificação e validação

Conforme expliquei no Capítulo 22, verificação e validação é o processo que demonstra que um programa atende a sua especificação (verificação) e atende às necessidades reais dos seus stakeholders (validação). As técnicas de verificação estáticas enfocam a análise manual ou automatizada do código-fonte do programa. A validação dinâmica ou teste tem a intenção de descobrir defeitos no programa ou demonstrar que o programa atende a seus requisitos. Quando a detecção de defeitos for o objetivo, o processo de testes poderá ser conduzido pelo conhecimento do código-fonte do programa. As métricas de cobertura de teste mostram a efetividade dos testes em causar a execução de declarações de código-fonte.

Para sistemas orientados a aspectos, os processos de testes de validação não são diferentes de qualquer outro sistema. O programa executável final é tratado como uma caixa-preta e os testes são aconselhados para mostrar se o sistema atende ou não a seus requisitos. Contudo, existem problemas reais com inspeções de programa quando aspectos são usados e com o uso de teste caixa-branca em que o código-fonte do programa é usado para identificar testes de defeitos potenciais. Além disso, como identificado por Katz (2005), o uso de aspectos introduz problemas adicionais para os testadores de programas:

1. Como os aspectos devem ser especificados de modo que os testes possam ser derivados para esses aspectos?
2. Como os aspectos podem ser testados independentemente do sistema-base com o qual eles devem ser compostos?
3. Como a interferência de aspectos pode ser testada? Conforme expliquei, a interferência de aspectos ocorre quando dois ou mais aspectos usam a mesma especificação de ponto de corte.
4. Como os testes podem ser projetados de modo que todos os pontos de junção do programa sejam executados e os testes de aspecto apropriados sejam aplicados?

Todas essas questões permanecem como problemas de pesquisa do desenvolvimento de software orientado a aspectos e, até o momento da escrita deste livro, não havia soluções gerais disponíveis. Fundamentalmente, os problemas de testes surgem porque os aspectos estão forte, em vez de fracamente, integrados com o código-base de um sistema. Eles são, portanto, difíceis de serem testados isoladamente. Por poderem ser compostos num programa em muitos lugares diferentes, você não pode assegurar que um aspecto que funciona com sucesso num ponto de junção funcionará em todos os pontos de junção. Não explico essas questões de teste além daqui. Em vez disso, enfoco como uma abordagem orientada a aspectos causa dificuldades para a inspeção de programas e testes de defeitos.

Inspeções de programa, conforme explicado no Capítulo 22, envolvem uma equipe de leitores que observam um código-fonte de um programa para descobrir defeitos que foram introduzidos pelo programador. É uma técnica muito efetiva de descoberta de defeitos. No entanto, programas orientados a aspectos não podem ser lidos seqüencialmente, de cima para baixo. São, portanto, mais difíceis de serem compreendidos pelas pessoas.

Uma diretriz geral para a facilidade de compreensão de programas é que um leitor deve ser capaz de ler um programa da esquerda para a direita, de cima para baixo, sem ter de alternar sua atenção para outras partes do código. Essa diretriz torna isso mais fácil para os leitores e também torna menos provável que os programadores cometerão erros quando a sua atenção estiver focada em uma única seção de código. O aumento da facilidade de leitura de programa foi a razão principal para a introdução de programação estruturada (Dijkstra, et al., 1972) e a eliminação de declarações de desvios incondicionais (*goto*) das linguagens de programação de alto nível.

Em um sistema orientado a aspectos, a leitura seqüencial de código é impossível. O leitor precisa examinar cada aspecto, compreender seus pontos de corte (que podem ser padrões) e o modelo de ponto de junção da linguagem orientada a aspectos. Ao ler o programa, ele ou ela deve então identificar muitos pontos de junção potenciais e direcionar a sua atenção para o código de aspecto para ver se ele pode ser composto naquele ponto. Sua atenção depois retorna ao fluxo de controle principal do código-base. Na realidade, isso é cognitivamente impossível e a única maneira possível de inspecionar um programa orientado a aspectos é por meio do uso de ferramentas de leitura de código.

Ferramentas de leitura de código que 'aplanem' um programa orientado a aspectos e apresentem um programa para o leitor com os aspectos 'compostos' dentro do programa nos pontos de junção especificados poderiam ser escritas. No entanto, o modelo de ponto de junção em uma linguagem pode ser dinâmico em vez de estático e pode ser impossível demonstrar que o programa aplanado se comportará exatamente da mesma maneira que o programa que será executado. Além disso, como é impossível para diferentes aspectos terem a mesma especificação de ponto de corte, a ferramenta de leitura de programa deve conhecer como o compositor de aspecto trata esses aspectos 'competidores' e como a composição será ordenada.

Conforme expliquei no Capítulo 23, caixa-branca ou teste estrutural é uma abordagem sistemática de teste na qual o conhecimento do código-fonte do programa é usado para projetar testes de defeitos. O objetivo é projetar testes que forneçam algum nível de cobertura de programa. Tipicamente, um conjunto de testes deve assegurar que cada desvio lógico em um programa seja seguido, sendo cada declaração do programa executada pelo menos uma vez. Os analisadores de execução de programa podem ser usados para demonstrar que esse nível de abrangência de teste foi atingido.

Em um sistema orientado a aspectos, há dois problemas com essa abordagem:

1. Como o conhecimento do código de programa pode ser usado para derivar sistematicamente os testes de programa?
2. O que significa exatamente cobertura de teste?

Para projetar testes em um programa estruturado (por exemplo, testes do código de um método) sem desvios incondicionais, você pode derivar um gráfico do fluxo do programa que revele todo caminho de execução lógica daquele programa. Você examina então o código e, para cada caminho por meio do gráfico de fluxo, escolhe os valores de entrada que causarão a execução daquele caminho.

No entanto, um programa orientado a aspectos não é um programa estruturado. O fluxo de controle é interrompido por declarações do tipo 'vem de' (Constantinos, et al., 2004). Em algum ponto de junção na execução do código-base, um aspecto pode ser executado. Não é claro para mim como o gráfico do fluxo de programa seria parecido em tal situação. É, portanto, difícil projetar testes de programa sistematicamente que assegurem que todas as combinações do código-base e os aspectos sejam executados.

Em um programa orientado a aspectos, há também o problema de decidir o que significa cobertura de testes. Significa que o código de cada aspecto é executado pelo menos uma vez? Essa é uma condição muito fraca por causa da interação entre aspectos e o código-base nos pontos de junção onde os aspectos são compostos. Portanto, a cobertura de testes significa que o código do aspecto deva ser executado pelo menos uma vez em todo ponto de junção especificado no ponto de corte do aspecto? O que acontece em situações em que aspectos diferentes definem o mesmo ponto de corte? Esses são tanto problemas teóricos quanto práticos? Necessitamos de ferramentas para dar suporte ao teste de programa orientado a aspectos que ajudarão a avaliar a extensão da cobertura de teste de um sistema.

Conforme explico no Capítulo 27, grandes projetos têm normalmente uma equipe de garantia de qualidade que estabelece padrões de teste e que requer garantia formal de que as revisões e o teste de programa foram completados de acordo com esses padrões. Os problemas de inspeção e derivação de testes para programas orientados a aspectos são barreiras significativas para a adoção do desenvolvimento de software orientado a aspectos em tais projetos de software de grande porte.

PONTOS-CHAVE

- O principal benefício da abordagem orientada a aspectos para desenvolvimento de software é que ela apóia a separação de assuntos. Com a representação de assuntos transversais como aspectos, esses assuntos podem ser compreendidos, reusados e modificados independentemente.
- O embaralhamento ocorre quando um módulo em um sistema inclui código que implementa requisitos diferentes de sistema. O fenômeno de espalhamento relacionado ocorre quando a implementação de um único assunto é espalhada por muitos componentes em um programa.
- Aspectos incluem um ponto de corte — uma declaração que define onde o aspecto será composto no programa e uma recomendação — o código para implementar o assunto transversal. Os pontos de junção são os eventos que podem ser referenciados em um ponto de corte.
- Para dar suporte à separação de assuntos, os sistemas podem ser projetados como um sistema central que implementa assuntos principais dos stakeholders e extensões que implementam assuntos secundários.
- Para identificar assuntos, você pode usar uma abordagem orientada a pontos de vista para que a engenharia de requisitos elicite os requisitos de stakeholders e identifique explicitamente os assuntos transversais de qualidade de serviço e de política.
- A transição de requisitos para o projeto pode ser feita pela identificação de casos de uso, sendo que cada caso de uso representa um assunto stakeholder. O projeto pode ser modelado usando uma versão estendida da UML com estereótipos de aspecto.
- Os problemas de inspeção e derivação de testes para programas orientados a aspectos são barreiras significativas para a adoção de desenvolvimento de software orientado a aspectos em projetos de software de grande porte.

LEITURAS SUGERIDAS

"Aspect-oriented programming". Esta edição especial do CACM tem uma série de artigos para um público geral, que são um bom ponto de partida para a leitura sobre programação orientada a aspectos. (*Comm. ACM*, 44 (10), outubro de 2001.)

Aspect-oriented software development. Livro de vários autores com grande variedade de artigos sobre desenvolvimento de software orientado a aspectos escritos por muitos pesquisadores importantes da área. (R. E. Filman, T. Elrad, S. Clarke and M. Aksit, Addison-Wesley, 2005.)

Aspect-oriented software development with use-cases. É um livro prático para projetistas de software. Os autores explicam como usar casos de uso para gerenciar a separação de assuntos e como usá-los como a base de um projeto orientado a aspectos. (I. Jacobsen e P. Ng, Addison-Wesley, 2005.)

EXERCÍCIOS

32.1 Quais são os diferentes tipos de assuntos de stakeholders que podem surgir em um sistema de grande porte? Como os aspectos podem dar suporte à implementação de cada um desses tipos de assuntos?

32.2 Resuma qual é o significado de embaralhamento e espalhamento. Usando exemplos, explique por que o embaralhamento e o espalhamento podem causar problemas quando mudam os requisitos de sistema.

32.3 Qual é a diferença entre um ponto de junção e um ponto de corte? Explique como eles facilitam a composição de código no programa para tratar de assuntos transversais.

32.4 Usando o sistema LIBSYS, explicado extensivamente neste livro, mostre como um projeto para esse sistema poderia ser organizado como um sistema central mais extensões. Qual funcionalidade seria incluída nas extensões?

32.5 Quais pontos de vista devem ser considerados quando se desenvolve uma especificação de requisitos de um sistema de registro de pacientes que recebem tratamento quanto à saúde mental, conforme explicado no Capítulo 30? Quais são provavelmente os assuntos transversais mais importantes?

32.6 Usando as funcionalidades delineadas para cada ponto de vista mostrado no Quadro 32.1, identifique seis outros casos de uso para o sistema de inventário de equipamentos, além dos mostrados na Figura 32.8. Quando apropriado, mostre como alguns deles poderiam ser organizados como casos de uso de extensão.

32.7 Usando a organização de estereótipo de aspecto ilustrada na Figura 32.12, desenvolva mais detalhadamente os aspectos Pedido e Monitor mostrados na Figura 32.11.

32.8 Explique como a interferência de aspectos pode surgir e sugira o que deve ser feito durante o processo de projeto de sistema para reduzir os problemas de interferência de aspectos.

32.9 Explique por que expressar as especificações de ponto de corte como padrões aumenta os problemas de teste e de debugging de programas orientados a aspectos. Para responder a isso, pense sobre como o teste de programa normalmente envolve a comparação entre saída esperada e real produzidas por um programa.

32.10 Sugira como você poderia usar aspectos para simplificar o debugging de programas.

GLOSSÁRIO

Ada
Linguagem de programação desenvolvida para o Departamento de Defesa dos Estados Unidos como um padrão de linguagem de desenvolvimento de software na área militar. Ela se baseia na pesquisa de linguagem de programação da década de 1970, inclui construções, como tipos abstratos de dados, e apóia a concorrência. Ela é ainda usada em sistemas militares e aeroespaciais complexos e de grande porte.

análise estática
Análise baseada em ferramentas de um código-fonte de programa para descobrir erros e anomalias. As anomalias, como atribuições sucessivas de uma variável, com nenhum uso intermediário, podem ser erros de programação.

aprimoramento de processo
A realização de mudanças em um processo com o propósito de torná-lo mais previsível ou aumentar a qualidade de suas saídas. Por exemplo, se seu propósito é reduzir o número de defeitos no software entregue, você poderia aprimorar o processo pela adição de novas atividades de validação.

arquitetura cliente-servidor
Modelo de arquitetura para sistemas distribuídos no qual a funcionalidade do sistema é oferecida como um conjunto de serviços fornecidos por um servidor. Esses serviços são acessados por computadores clientes que fazem uso deles. As variantes dessa abordagem, como a arquitetura cliente-servidor de três camadas, usam vários servidores.

arquitetura de referência
Arquitetura de sistema genérica e idealizada que inclui todas as características que os sistemas podem incorporar. Essa é uma maneira de informar aos projetistas sobre a estrutura geral dessa classe de sistema.

arquitetura de software
Modelo da estrutura fundamental e da organização de um sistema de software.

C
Linguagem de programação originalmente desenvolvida para auxiliar a implementação do sistema UNIX. C é uma linguagem de implementação de sistema relativamente de baixo nível que permite acessar o hardware do sistema e que pode ser compilada para um código eficiente. Ela é ainda amplamente usada para programação de sistemas de baixo nível.

C++
Linguagem de programação orientada a objetos que é um superconjunto de C.

caso de confiabilidade
Documento estruturado usado para registrar reivindicações feitas por um desenvolvedor de sistema quanto à confiabilidade de um sistema.

caso de segurança
Argumento estruturado de que um sistema é seguro. Geralmente exigido por órgãos regulamentadores, como os de segurança nuclear.

caso de uso
A especificação de um tipo de interação com um sistema.

cenário
Descrição da maneira típica na qual um sistema é usado ou um usuário realizou alguma atividade.

ciclo de vida de software
Freqüentemente usado com outro nome para o processo de software. Originalmente criado para se referir ao modelo em cascata do processo de software.

classe de objeto
Classe de objeto define os atributos e operações de objetos. Os objetos são criados em tempo de execução pela criação de instâncias de definição de classe. O nome da classe de objeto pode ser usado como um nome de tipo em algumas linguagens orientadas a objetos.

CMMI
Abordagem integrada para a modelagem da maturidade de capacitação de processo. Ela apóia a modelagem de maturidade discreta e contínua, além de integrar modelos de maturidade de processos de engenharia de sistemas e de software.

código de ética e prática profissional
Conjunto de diretrizes que estabelecem a ética e o comportamento profissional esperado dos engenheiros de software. Isso foi definido pelas principais sociedades de profissionais dos EUA (ACM e IEEE) e define o comportamento ético em oito tópicos: público, cliente e empregador, produto, julgamento, gerenciamento, colegas, profissão e a própria pessoa.

COM+
Modelo de componente projetado para uso em plataformas Microsoft.

Common Request Broker Architecture (CORBA)
Conjunto de padrões propostos pelo OMG que define um modelo de objetos distribuídos e as comunicações de objetos.

componente
Unidade de software independente que pode ser implantada, completamente definida e acessada por meio de um conjunto de interfaces.

Computer-Aided Software Engineering (CASE)
Processo de desenvolvimento de software que usa apoio automatizado.

confiabilidade
A capacidade de um sistema de entregar serviços conforme especificado. A confiabilidade pode ser especificada quantitativamente como a probabilidade de falha sob demanda ou como uma taxa de ocorrência de falhas.

confidencialidade
A confidencialidade de um sistema é uma propriedade agregada que leva em consideração a segurança, a confiabilidade, a disponibilidade, a proteção e outros atributos do sistema. Ela reflete o quanto seus usuários podem confiar nele.

construção de sistema
O processo de compilação de componentes ou unidades que constituem o sistema e a ligação deles com outros componentes para criar um programa executável. A construção do sistema normalmente é automatizada de modo que a recompilação é minimizada. Essa automação pode estar incorporada ao sistema de processamento de linguagem (como em JAVA) ou pode envolver ferramentas CASE para apoiar a construção do sistema.

controle da qualidade (QC – Controle da Qualidade)
O processo de assegurar que uma equipe de desenvolvimento de software esteja seguindo os padrões de qualidade.

desenvolvimento de software orientado a aspectos
Abordagem para o desenvolvimento de software que combina o desenvolvimento baseado em geradores e o orientado a componentes. Os assuntos transversais são identificados em um programa e a implementação desses assuntos é definida como aspectos. Um compositor de programa compõe, então, os aspectos em locais apropriados no programa.

desenvolvimento incremental
Abordagem para o desenvolvimento de software no qual este é entregue e implantado em incrementos.

desenvolvimento iterativo
Abordagem para o desenvolvimento de software na qual os processos de especificação, projeto, programação e teste são intercalados.

desenvolvimento orientado a objetos (OO)
Abordagem para o desenvolvimento de software na qual as abstrações fundamentais do sistema são objetos independentes. O mesmo tipo de abstração é usado durante a especificação, o projeto e o desenvolvimento.

desenvolvimento rápido de aplicação (RAD – Rapid Application Development)
Abordagem para o desenvolvimento de software proposta para a entrega rápida do software. Ela envolve, freqüentemente, o uso de programação de banco de dados e ferramentas de apoio ao desenvolvimento, como geradores de telas e relatórios.

detecção de defeitos
O uso de processos e verificação run-time para detectar e remover defeitos em um programa antes que esses defeitos resultem em uma falha do sistema.

diagrama de atividades (PERT)
Diagrama usado pelos gerentes de projeto para demonstrar as dependências entre as tarefas que devem ser concluídas. O diagrama mostra as tarefas, o tempo esperado para concluir essas tarefas e suas dependências. O caminho crítico é o caminho mais longo (em termos de tempo necessário para concluir as tarefas) por meio do diagrama de atividades. Ele define o tempo mínimo necessário para concluir o projeto.

diagrama de barras (Gantt)
Diagrama usado pelos gerentes de projeto para demonstrar as tarefas, o cronograma associado a essas tarefas e as pessoas que trabalharão nelas. Ele mostra o início das tarefas e suas datas finais, além das alocações de pessoal *versus* uma linha de tempo.

diagrama de seqüência
Diagrama que mostra a seqüência de interações necessárias para concluir alguma operação. Na UML, os diagramas de seqüência podem ser associados a casos de uso.

dinâmica de evolução de programa
O estudo de meios pelos quais um desenvolvimento de sistema de software muda.

disponibilidade
A prontidão de um sistema para fornecer serviços quando solicitado. A disponibilidade geralmente é expressa como um número decimal, de modo que uma disponibilidade de 0,999 significa que o sistema pode fornecer serviços por 999 de 1000 unidades de tempo.

domínio
Problema ou área de negócio específica no qual os sistemas de software são usados. Exemplos de domínios são: controle de tempo real, processamento de dados de negócios e comutação de telecomunicações.

engenharia de sistemas
Processo relacionado à especificação de um sistema, integração de seus componentes e teste para verificar se o sistema atende a seus requisitos. A engenharia de sistema está relacionada a todo o sistema sociotécnico — software, hardware e processos operacionais — não apenas ao software do sistema.

engenharia de software baseada em componenentes (CBSE – component-based software engineering)
O desenvolvimento de software pela composição de componentes independentes e que podem ser implantados.

engenharia de software Cleanroom
Abordagem para desenvolvimento de software na qual o propósito é evitar a introdução de defeitos no software (pela analogia com uma sala limpa usada na fabricação de semicondutores). O processo envolve a especificação formal de

software, transformação estruturada de uma especificação para um programa, o desenvolvimento de argumentos de correção e testes estatísticos de programa.

enterprise Java beans (EJB)
Modelo de componente baseado em JAVA.

especificação formal, algébrica
Método de especificação matemática no qual um sistema ou componente é especificado pela definição de relacionamentos entre as operações definidas em suas interfaces externas.

especificação formal, baseada em modelos
Método de especificação matemática no qual um sistema ou componente é especificado pela definição de pré-condições, pós-condições e invariantes que se aplicam ao estado do sistema.

etnografia
Técnica de observação que pode ser usada na elucidação e análise de requisitos. O etnógrafo se integra no ambiente do usuário e observa seus hábitos cotidianos de trabalho. Requisitos para apoio de software podem ser inferidos a partir dessas observações.

extreme programming (XP)
Método ágil de desenvolvimento de software que inclui práticas, como requisitos baseados em cenários, desenvolvimento *test-first* e programação em pares.

família de aplicações
Conjunto de programas de aplicação de software que possuem uma arquitetura comum e funcionalidades genéricas. Esses programas podem ser configurados para as necessidades de clientes específicos pela modificação de componentes e parâmetros de programa.

ferramenta CASE
Ferramenta de software, como um editor de projeto ou um debugger (depurador) de programa, usado para apoiar uma atividade no processo de desenvolvimento de software.

framework de aplicação
Estrutura genérica em um domínio específico que pode formar a base de uma família de aplicações. Os frameworks de aplicações são geralmente implementados como um conjunto de classes concretas e abstratas, especializadas e instanciadas para criar uma aplicação.

garantia da qualidade (QA – Quality Assurance)
O processo geral de definição de como a qualidade de software pode ser atingida e como a organização de desenvolvimento sabe que o software tem o nível de qualidade necessário.

gerador de programa
Programa que gera um outro programa a partir de uma especificação abstrata de alto nível. O gerador incorpora conhecimento reusado em cada atividade de geração.

gerenciamento de configurações
Processo de gerenciamento de mudanças de um produto de software em desenvolvimento. O gerenciamento de configurações envolve o planejamento de configurações, o gerenciamento de versões, a construção do sistema e o gerenciamento de mudanças.

gerenciamento de requisitos
Processo de gerenciamento de mudanças de requisitos para assegurar que as mudanças feitas são adequadamente analisadas e acompanhadas ao longo do sistema.

gerenciamento de riscos
Processo de identificação de riscos, avaliação de sua gravidade, planejamento de medidas para verificar se os riscos aumentam e monitoração de riscos e do processo de software.

inspeção de programa
Processo de verificação no qual um grupo de inspetores examina um programa, linha por linha, com o propósito de detectar erros de programa.

Interface de Programa de Aplicação (API – Application Program Interface)
Interface geralmente especificada como um conjunto de operações definida por um programa de aplicação que permite acesso à funcionalidade do programa. Isso significa que essa funcionalidade pode ser chamada diretamente por outros programas e não acessada somente por meio da interface com o usuário.

interface
Especificação de atributos e operações associadas a um componente de software. A interface é usada como meio de acesso à funcionalidade do componente.

ISO 9000
Padrão para o processo de gerenciamento de qualidade definido pela International Standards Organisation (ISO).

item de configuração
Unidade que pode ser lida por uma máquina, como um documento ou um arquivo de código-fonte, sujeita a mudanças e sendo a mudança controlada por um sistema de gerenciamento de configurações.

Java
Linguagem de programação orientada a objetos projetada pela Sun para ser independente de plataforma.

linguagem de restrição de objetos (OCL – Object Constraint Language)
Linguagem parte da UML, usada para definir predicados que se aplicam a classes de objeto e interações em um modelo de UML.

linha de produtos de software
Veja família de aplicações.

manutenção
Processo de realizar mudanças em um sistema depois que ele foi colocado em operação.

método estruturado
Método de projeto de software que define os modelos de sistema a serem desenvolvidos, as regras e as diretrizes que devem ser aplicadas a esses modelos e um processo a ser seguido no desenvolvimento do projeto.

métodos ágeis
Métodos de desenvolvimento de software voltados para entrega rápida. O software é desenvolvido e entregue em incrementos, e a documentação e a burocracia do processo são minimizados.

métodos formais
Métodos de desenvolvimento de software baseados em abordagens matematicamente rigorosas e que modelam o software usando construções matemáticas formais, como predicados e conjuntos.

métrica de software
Atributo de sistema de software ou processo que pode ser expresso e medido numericamente. Métricas de processo são atributos do processo, como o tempo necessário para concluir uma tarefa; métricas de produto são atributos do software em si, como o tamanho ou a complexidade.

middleware
Uma infra-estrutura de software em um sistema distribuído. Ela ajuda a gerenciar as interações entre as entidades distribuídas no sistema e o banco de dados do sistema. Exemplos de middleware são um localizador de objetos e um sistema de gerenciamento de transações.

modelagem algorítmica de custos
Abordagem para a estimativa de custo de software na qual uma fórmula é usada para estimar o custo do projeto. Os parâmetros na fórmula são atributos a partir do projeto e do software em si.

Modelagem Construtiva de Custos (COCOMO – Constructive Cost Modelling)
Talvez o modelo de estimativa algorítmica de custos mais conhecido.

modelagem de crescimento de confiabilidade
O desenvolvimento de um modelo de como a confiabilidade de um sistema muda (com esperança de melhora) à medida que ele é testado e os defeitos de programas são removidos.

modelo de componente CORBA
Modelo de componente projetado para uso da plataforma CORBA.

modelo de componente
Conjunto de padrões para a implementação, documentação e implantação de componentes. Esses padrões abordam as interfaces específicas que podem ser fornecidas por um componente, denominação, interoperação e composição de componentes. Os modelos de componentes fornecem a base para o middleware que apóia a execução de componentes.

modelo de domínio
Definição de abstrações de domínio, como as políticas, procedimentos, objetos, relacionamentos e eventos. Ele serve como base de conhecimento de alguma área de problema.

Modelo de Maturidade de Capacitação de Pessoal (P-CMM – People Capability Maturity Model)
Modelo de maturidade de processo que reflete o grau de eficiência de uma organização no gerenciamento de habilidades, treinamento e experiência das pessoas nessa organização.

modelo de maturidade de processo
Modelo extenso no qual um processo inclui boas práticas e capacidades de ponderação e medição conduzidas para o aprimoramento do processo.

modelo de objeto
Modelo de sistema de software estruturado e organizado como um conjunto de classes de objeto e os relacionamentos entre essas classes. Pode haver várias perspectivas diferentes do modelo, como uma perspectiva de estado e uma perspectiva de seqüência.

modelo de processo
Representação abstrata de um processo. Os modelos de processo podem ser desenvolvidos a partir de várias perspectivas e podem demonstrar as atividades envolvidas em um processo, os artefatos usados, as restrições que se aplicam a ele e os papéis das pessoas que o aprovam.

modelo em cascata
Modelo de processo de software no qual existem estágios discretos de desenvolvimento: especificação, projeto, implementação, teste e manutenção. Em princípio, um estágio deve ser concluído antes que o avanço para o próximo estágio seja possível. Na prática, existe iteração entre estágios.

modelo em espiral
Modelo de um processo de desenvolvimento no qual o processo é representado por uma espiral. Cada volta da espiral incorpora os diferentes estágios do processo. À medida que se passa de uma volta da espiral para outra, repetem-se todos os estágios do processo.

Object Management Group (OMG)
Grupo de empresas formado para desenvolver padrões para o desenvolvimento orientado a objetos. Exemplos de padrões promovidos pela OMG são CORBA, UML e MDA.

ocultação de informações
O uso de construtores da linguagem de programação para ocultar a representação de estruturas de dados e para controlar acessos externos a essas estruturas.

padrão de projeto
Solução aprovada para um problema comum que capta a experiência e as boas práticas de modo que possam ser usadas. Ela é uma representação abstrata que pode ser instanciada de várias maneiras.

plano de qualidade
Plano que define os processos e os procedimentos de qualidade a serem usados. Ele envolve a seleção e a instanciação de padrões para produtos e processos e a definição de atributos necessários de qualidade do sistema.

prevenção de defeitos
Desenvolvimento de software de modo que não sejam introduzidos defeitos nesse software.

princípios de projeto de interface com o usuário
Conjunto de princípios que incorporam boas práticas para o projeto de interface com o usuário.

processo de software
Conjunto de atividades e processos relacionados envolvidos no desenvolvimento e evolução de um sistema de software.

projeto de interface com o usuário
O processo de projetar o modo como os usuários acessam a funcionalidade do sistema e como as informações produzidas pelo sistema são exibidas.

propriedade emergente
Propriedade que se torna aparente somente depois que todos os componentes tenham sido integrados para criar o sistema.

proteção
A capacidade de um sistema de proteger-se contra a invasão acidental ou deliberada.

prototipação Mágico de Oz
Abordagem para a prototipação de interface com o usuário na qual os comandos de entrada de um usuário são interpretados por uma pessoa que responde como se fosse o computador.

Rational Unified Process (RUP)
Modelo genérico de processo de software que apresenta o desenvolvimento de software como uma atividade iterativa de quatro fases: concepção, elaboração, construção e transição. A concepção estabelece um caso de negócio para o sistema, a elaboração define a arquitetura, a construção implementa o sistema e a transição implanta o sistema no ambiente do cliente.

reengenharia
Modificação de um sistema de software para torná-lo mais fácil de compreender e mudar. A reengenharia envolve, freqüentemente, reestruturação e organização de software e dados, simplificação de programas e nova documentação.

reengenharia, processo de negócio
Mudança de um processo de negócio para atender a algum novo objetivo organizacional, como redução de custo e execução mais rápida.

release
Versão de um sistema de software disponibilizada aos clientes do sistema.

requisito de confiabilidade
Requisito de sistema incluído para auxiliar a atingir a confiabilidade necessária para um sistema. Requisitos de confiabilidade não funcionais especificam valores de atributos de confiabilidade; requisitos de confiabilidade funcional são requisitos funcionais para evitar, detectar, tolerar ou recuperar-se de defeitos e falhas do sistema.

requisito, funcional
Declaração de alguma função ou característica que deve ser implementada em um sistema.

requisito, não-funcional
Uma declaração de uma restrição ou comportamento esperado que se aplica a um sistema. Essa restrição pode se referir às propriedades emergentes do software sendo desenvolvido ou ao processo de desenvolvimento.

risco
Saída indesejável que apresenta uma ameaça ao alcance de algum objetivo. Um risco de processo ameaça o cronograma ou o custo de um processo; um risco de produto que pode significar que algum dos requisitos do sistema pode não ser atingido.

segurança
A capacidade de um sistema funcionar sem falhas catastróficas.

serviço Web
Componente de software independente que pode ser acessado por meio da Internet usando protocolos padrões. O SOAP (Standard Object Access Protocol) é usado para a troca de informações de serviço Web. A WSDL (Web Service Definition Language) é usada para definir as interfaces de serviço Web.

servidor
Programa que fornece algum serviço a outros (cliente) programas.

sistema crítico
Sistema de computador cuja falha pode resultar em significativos prejuízos econômicos, humanos ou ambientais.

sistema de objetos distribuídos
Sistema distribuído no qual os componentes executáveis são objetos.

sistema de processamento de dados
Sistema cujo propósito é processar uma quantidade grande de dados estruturados. Esses sistemas geralmente processam os dados em lotes e seguem um modelo entrada-processo-saída. Exemplos de sistemas de processamento de dados são sistemas de cobrança e faturamento, além de sistemas de pagamento.

sistema de processamento de linguagem
Um sistema que converte uma linguagem em outra. Por exemplo, um compilador é um sistema de processamento de linguagens que converte o código-fonte do programa no código-objeto.

sistema de processamento de transações
Sistema que assegura que as transações são processadas de modo que não interfiram umas com as outras e, assim, uma falha de transação individual não afeta outras transações ou dados do sistema.

sistema de tempo real
Sistema que deve responder e processar eventos externos em 'tempo real'. A correção do sistema não depende somente do que ele faz, mas também de quão rapidamente ele faz. Sistemas de tempo real são, geralmente, organizados como um conjunto de processos seqüenciais que cooperam entre si.

sistema distribuído
Sistema de software no qual os subsistemas ou componentes operam em diferentes processadores.

sistema legado
Um sistema sociotécnico útil ou essencial para uma organização, mas que foi desenvolvido usando tecnologia ou métodos obsoletos. Devido aos sistemas legados freqüentemente realizarem funções críticas de negócio, eles devem ser mantidos.

sistema ponto a ponto
Um sistema distribuído no qual não há distinção entre clientes e servidores. Os computadores no sistema podem agir como clientes e servidores. Aplicações ponto a ponto incluem sistemas de compartilhamento de arquivos, de mensagens instantâneas e de apoio à cooperação.

sistema sociotécnico
Um sistema, incluindo os componentes de hardware e software, que define os processos operacionais seguidos pelos operadores humanos e que funciona dentro de uma organização. Ele é, portanto, influenciado por políticas, procedimentos e estruturas organizacionais.

sistemas baseados em eventos
Sistemas em que o controle da operação é determinado por eventos gerados no ambiente do sistema. A maior parte dos sistemas de tempo real são sistemas baseados em eventos.

Structured Query Language (SQL)
Uma linguagem-padrão usada para a programação de banco de dados relacional.

tipo abstrato de dados
Tipo cuja representação é ocultada e que está definida por suas operações.

tolerância a defeitos
A capacidade de um sistema de continuar em execução mesmo depois da ocorrência de defeitos.

transação
Uma unidade de interação com um sistema de computador. As transações são independentes e atômicas (elas não são decompostas em unidades menores), além de constituírem uma unidade fundamental de recuperação, consistência e concorrência.

Unified Modeling Language (UML)
Uma linguagem gráfica usada no desenvolvimento orientado a objetos, que inclui diversos tipos de modelos que fornecem diferentes visões de um sistema. A UML tornou-se um padrão de fato para a modelagem orientada a objetos.

validação
Processo de verificação de se um sistema atende às necessidades e expectativas do cliente.

verificação
O processo de verificação que avalia se um sistema atende à sua especificação.

workbench CASE
Um conjunto integrado de ferramentas CASE que trabalham em conjunto para apoiar uma atividade principal de processo, como o projeto de software ou o gerenciamento de configurações.

XML
Extended Markup Language. XML é uma liguagem de marcação de texto que apóia a troca de dados estruturados. Cada campo de dado é delimitado pelas marcas que fornecem informações sobre esse campo. O XML é amplamente usado e tornou-se a base de protocolos para serviços Web.

Z
Uma linguagem de especificação formal baseada em modelos desenvolvida na Universidade de Oxford na Inglaterra.

REFERÊNCIAS

Abbott, R. (1983). "Program design by informal English descriptions". *Comm. ACM,* **26**(11), 882-94. (Cap. 14)

Abdel-Ghaly, A. A., Chan, P. Y. et al. (1986). "Evaluation of competing software reliability predictions". *IEEE trans. on software engineering*, **SE-12**(9), 950-67. (Cap. 24)

Ackroyd, S., Harper, R. e*t al*. (1992). *Information technology and practical police work.* Milton Keynes: Open University Press. (Cap. 2)

Adams, E. N. (1984). "Optimizing preventative service of software products". *IBM j. res. & dev.*, **28**(1), 2-14. (Cap. 3)

Ahern, D. M., Clouse, A. et al. (2001). *CMMI distilled.* Reading, MA: Addison-Wesley. (Caps. 28, 29)

Albrecht, A. J. (1979). "Measuring application development productivity". *Proc. SHARE/ GUIDE IBM application development symposium*. (Cap. 26)

Albrecht, A. J. e Gaffney, J. E. (1983). "Software function, lines of code and development effort prediction: a software science validation". *IEEE trans. on software engineering*, **SE-9**(6), 639-47. (Cap. 26)

Alexander, C., Ishikawa, S. et al. (1977). *A pattern language.* Oxford: Oxford University Press. (Cap. 18)

Ambler, S. W. e Jeffries, R. (2002). *Agile modeling*. Nova York: John Wiley & Sons. (Cap. 17)

Andrews, T. e*t al.* (2003). "Business process execution language for Web services". www.128.ibm.com/developerworks/library/ws-bpel/. (Cap. 31)

Anderson, R. (2001). *Security engineering: a guide to building dependable distributed systems.* Chichester: John Wiley & Sons. (Caps. 24, 30)

Appelrath, H.-J. e Ritter, J. (2000). *SAP R/3 implementation: methods and tools (SAP excellence).* Berlin: Springer-Verlag. (Cap. 13)

Armour, P. (2002). "Ten unmyths of project estimation". *Comm. ACM*, **45**(11), 15-18. (Cap. 26)

Aron, J. D. (1974). *The program development process.* Reading, MA: Addison-Wesley. (Cap. 25)

Arthur, L. J. (1988). *Software evolution.* Nova York: John Wiley & Sons. (Cap. 21)

Avizienis, A. (1985). "The N-version approach to fault-tolerant software". *IEEE Trans. on software engineering*, **SE-11**(12), 1491-501. (Cap. 20)

Avizienis, A. A. (1995). "A methodology of N-version programming". In *Software fault tolerance* (M. R. Lyu, ed.). Nova York: John Wiley & Sons, 23-46. (Cap. 20)

Bagert, D. J. (2002). "Texas licensing of software engineers: all's quiet for now". *Comm. ACM*, **45**(11), 92-4. (Cap. 24)

Baker, F. T. (1972). "Chief programmer team management of production programming". *IBM systems j.*, **11**(1), 56-73. (Cap. 25)

Baker, T. (2002). "Lessons learned integrating COTS into systems". *Proc. ICCBSS 2002 (1st int. conf on COTS-based software systems)*, Orlando, FL: Springer-Verlag. (Cap. 18)

Balk, L. D. e Kedia, A. (2000). "PPT: a COTS integration case study". *Proc. int. conf. on software engineering*, Limerick, Ireland: ACM Press. (Cap. 18)

Bamford, R. e Deibler, W. J., eds. (2003). "ISO 9001: 2000 for software and systems providers: an engineering approach". CRC Press. (Cap. 27)

Banker, R. D., Datar, S. M. et al. (1993). "Software complexity and maintenance costs". *Comm. ACM*, **36**(11), 81-94. (Caps. 21, 26)

Banker, R., Kauffman, R. et al. (1994). "An empirical test of object-based output measurement metrics in a computer-aided software engineering (CASE) environment". *J. of management info. sys.*, **8**(3), 127-50. (Cap. 26)

Bansler, J. P. e Bødker, K. (1993). "A reappraisal of structured analysis: design in an organizational context". *ACM trans. on information systems*, **11**(2), 165-93. (Cap. 4)

Barker, R. (1989). *CASE* method: entity relationship modelling.* Wokingham: Addison-Wesley. (Cap. 8)

Barnard, J. e Price, A. (1994). "Managing code inspection information". *IEEE software*, **11**(2), 59-69. (Caps. 22, 27)

Basili, V. e Green, S. (1993). "Software process improvement at the SEL". *IEEE software*, **11**(4), 58-66. (Cap. 28)

Basili, V. R. e Rombach, H. D. (1988). "The TAME project: towards improvement-oriented software environments". *IEEE trans. on software engineering*, **14**(6), 758-73. (Caps. 27, 28)

Bass, B. M. e Dunteman, G. (1963). "Behaviour in groups as a function of self, interaction and task orientation". *J. abnorm. soc. psychology*, **66**(4), 19-28. (Cap. 25)

Bass, L., Clements, P. et al. (2003). *Software architecture in practice, 2nd edn.* Boston: Addison-Wesley. (Cap. 11)

Baumer, D., Gryczan, G. et al. (1997). "Framework development for large systems". *Comm. ACM*, **40**(10), 52-9. (Cap. 18)

Beck, K. (1999). "Embracing change with extreme programming". *IEEE computer*, **32**(10), 70-8. (Caps. 6, 17)

Beck, K. (2000). *Extreme programming explained.* Boston: Addison-Wesley. (Caps. 4, 17, 25, 26)

Beck, K. e Cunningham, W. (1989). "A laboratory for teaching object-oriented thinking". *Proc. OOPSLA'89*, New Orleans: ACM Press. (Cap. 14)

Bentley, R., Rodden, T. et al. (1992). Ethnographically informed systems design for air traffic control. *Proc. CSCW'92*, Toronto: ACM Press. (Cap. 16)

Berczuk, S. P. e Appleton, B. (2002). *Software configuration management patterns: effective teamwork, practical integration.* Boston: Addison-Wesley. (Cap. 18)

Berghel, H. (2001). "The code red worm". *Comm. ACM*, **44**(12), 15-19. (Caps. 3, 30)

Berliner, B. (1990). "CVS II: parallelizing software development". *Proc. 1990 winter USENIX conference*, Washington, DC: USENIX Assoc. (Cap. 29)

Bernstein, P. A. (1996). "Middleware: a model for distributed system services". *Comm. ACM*, **39**(2), 86-97. (Cap. 12)

Bersoff, E. H. e Davis, A. M. (1991). "Impact of life cycle models on software configuration management". *Comm. ACM*, **34**(8), 104-18. (Cap. 29)

Bezier, B. (1990). *Software testing techniques, 2nd edn.* Nova York: Van Nostrand Rheinhold. (Cap. 23)

Biggerstaff, T. (1998). "A perspective of generative reuse". *Annals of software engineering*, **5**, 169-226. (Cap. 18)

Birrer, I., Pasetti, A. e*t al.* (2005). "The XWeaver Project: aspect-oriented programming for on-board applications". http://control.ee.ethz.ch/index.cgi?page=publications;action= details;id=2361.

Bishop, M. (2005). *Introduction to computer security*, Boston, MA: Addison-Wesley. (Cap. 30)

Bishop, P. e Bloomfield, R. E. (1995). "The SHIP safety case approach". *Proc. Safecomp'95*, Belgirate, Italy: Springer-Verlag. (Cap. 24)

Bishop, P. e Bloomfield, R. E. (1998). "A methodology for safety case development". *Proc. safety-critical systems symposium*, Birmingham, UK: Springer-Verlag. (Cap. 24)

Blevins, D. (2001). "Overview of the Enterprise Java Beans component model". In *Component-based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 589-606 (Cap. 19)

Boehm, B. (1997). *COCOMO II model definition manual.* Center for Software Engineering, Univ. of Southern California (http://sunset.usc.edu/research/COCOMOII). (Cap. 26)

Boehm, B. e Abts, C. (1999). "COTS integration: plug and pray?" *IEEE computer*, **32**(1), 135-8. (Cap. 18)

Boehm, B. e Royce, W. (1989). "Ada COCOMO and the Ada process model". *Proc. 5th COCOMO users' group meeting*, Pittsburgh: Software Engineering Institute. (Cap. 26)

Boehm, B. W. (1979). "Software engineering; R & D trends and defense needs". In *Research directions in software technology* (P. Wegner, ed.). Cambridge, MA: MIT Press, 1-9. (Cap. 22)

Boehm, B. W. (1981). *Software engineering economics.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 26)

Boehm, B. W. (1988). "A spiral model of software development and enhancement". *IEEE computer*, **21**(5), 61-72. (Caps. 4, 5)

Boehm, B. W., Abts, C. et al. (2000). *Software cost estimation with COCOMO II.* Upper Saddle River, NJ: Prentice Hall. (Caps. 19, 22, 26)

Boehm, B. W., McClean, R. L. et al. (1975). "Some experience with automated aids to the design of large-scale reliable software". *IEEE trans. on software engineering*, **SE-1**(1), 125-33. (Cap. 3)

Boehm, B., Clark, B. et al. (1995). "Cost models for future life cycle processes: COCOMO 2". *Annals of software engineering*, **1**, 57-94. (Cap. 26)

Bolognesi, T. e Brinksma, E. (1987). "Introduction to the ISO specification language LOTOS". *Computer networks*, **14**(1), 25-59. (Cap. 10)

Booch, G. (1987). *Software components with Ada: structures, tools, and subsystems.* Menlo Park, CA: Benjamin-Cummings. (Cap. 18)

Booch, G. (1994). *Object-oriented analysis and design with applications.* Redwood City, CA: Benjamin-Cummings. (Caps. 1, 4, 8, 14)

Booch, G., Rumbaugh, J. et al. (1999). *The unified modeling language user guide.* Reading, MA: Addison-Wesley. (Caps. 1, 4, 8)

Borchers, J. (2001). *A pattern approach to interaction design.* Nova York: John Wiley & Sons. (Cap. 18)

Bosch, J. (2000). *Design and use of software architectures.* Harlow: Addison-Wesley. (Cap. 11)

Bourne, S. R. (1978). "The Unix shell". *Bell sys. tech. j.*, **57**(6), 1971-90. (Cap. 17)

Bracket, M. H. (1994). *Data sharing using a common data architecture.* Nova York: John Wiley & Sons. (Cap. 13)

Brazendale, J. e Bell, R. (1994). "Safety-related control and protection systems: standards update". *IEEE computing and control engineering j.*, **5**(1), 6-12. (Cap. 9)

Brilliant, S. S., Knight, J. C. et al. (1990). "Analysis of faults in an N-version software experiment". *IEEE trans. on software engineering*, **16**(2), 238-47. (Cap. 20)

Brinch-Hansen, P. (1973). *Operating system principles.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 15)

Brooks F. P. (1975). *The mythical man month.* Reading, MA: Addison-Wesley. (Cap. 25)

Brown, A. W., Earl, A. N. et al. (1992). *Software engineering environments.* Londres: McGraw-Hill. (Cap. 11)

Budgen, D. (2003). *Software design, 2nd edn.* Harlow: Addison-Wesley. (Cap. 8)

Burns, A. e Wellings, A. (2001). *Real-time systems and programming languages.* Harlow: Addison-Wesley. (Cap. 15)

Butler, H. (1994). "Virtual remote: the centralized expert". *HP journal*, **45**(5), http://www.hpl.hp.com/hpjournal/94oct/oct94a13.htm. (Cap. 18)

Buxton, J. (1980). *Requirements for Ada programming support environments: stoneman.* Washington, DC: US Department of Defense. (Cap. 11)

Cabrera, L. F., Copeland, G. e*t al.* (2005). "Web services coordination" (WS-Coordination). ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf. (Cap. 31)

Checkland, P. (1981). *Systems thinking, systems practice.* Chichester: John Wiley & Sons. (Cap. 2)

Checkland, P. e Scholes, J. (1990). *Soft systems methodology in action.* Chichester: John Wiley & Sons. (Cap. 2)

Chen, P. (1976). "The entity relationship model — towards a unified view of data". *ACM trans. on database systems*, **1**(1), 9-36. (Cap. 8)

Chess, B. e Mcgraw, G. (2004). "Static analysis for security". *IEEE security and privacy*, **2**(6), 76-9. (Cap. 30)

Chidamber, S. e Kemerer, C. (1994). "A metrics suite for object-oriented design". *IEEE trans. on software engineering*, **20**(6), 476-93. (Cap. 27)

Chikofsky, E. J. e Cross, J. H. (1990). "Reverse engineering and design recovery: a taxonomy". *IEEE software*, **7**(1), 13-17. (Cap. 21)

Clark, S. e Baniassad, E. (2005). *Aspect-oriented analysis and design: the theme approach*, Harlow, UK: Addison Wesley.

Clements, P., Bachmann, F. et al. (2002). *Documenting software architectures: views and beyond.* Boston: Addison-Wesley. (Cap. 11)

Coad, P. e Yourdon, E. (1990). *Object-oriented analysis.* Englewood Cliffs, NJ: Prentice Hall. (Caps. 8, 14)

Cobb, R. H. e Mills, H. D. (1990). "Engineering software under statistical quality control". *IEEE software*, **7**(6), 44-54. (Cap. 22)

Cockburn, A. (2001). *Agile software development.* Reading, MA: Addison-Wesley. (Cap. 17)

Codd, E. F. (1979). "Extending the database relational model to capture more meaning". *ACM trans. on database systems*, **4**(4), 397-434. (Cap. 8)

Cohen, B., Harwood, W. T. et al. (1986). *The specification of complex systems.* Wokingham: Addison-Wesley. (Cap. 10)

Colyer, A. e Clement, A. (2005). "Aspect-oriented programming with AspectJ". *IBM systems j.*, **44**(2), 301-8.

Colyer, A., Clement, A. e*t al.* (2005). *Eclipse AspectJ*, Upper Saddle River, NJ: Addison Wesley.

Constantine, L. L. e Yourdon, E. (1979). *Structured design.* Englewood Cliffs, NJ: Prentice Hall. (Caps. 4, 8)

Constantinos, C., Skotiniotis, T. e*t al.* (2004). "AOP considered harmful". *Proc. european interactive workshop on aspects in software (EIWAS'04)*, Berlin, Germany. www.infosun.fmi.uni-passau.de/st/papers/EIWAS04/stoerzer04aop_harmful.pdf.

Cooling, J. (2003). *Software engineering for real-time systems.* Harlow: Addison-Wesley. (Cap. 15)

Coulouris, G., Dollimore, J. et al. (2001). *Distributed systems: concepts and design.* Harlow: Addison-Wesley. (Cap. 12)

Councill, W. T. e Heineman, G. T. (2001). "Definition of a software component and its elements". In *Component-based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 5-20. (Cap. 19)

Crabtree, A. (2003). *Designing collaborative systems: a practical guide to ethnography.* Londres: Springer. (Cap. 16)

Crosby, P. (1979). *Quality is free.* Nova York: McGraw-Hill. (Cap. 27)

Curtis, B., Hefley, W. E. et al. (2001). *The people capability model: guidelines for improving the workforce.* Boston: Addison-Wesley. (Caps. 25, 28)

Cusamano, M. (1989). "The software factory: a historical interpretation". *IEEE software*, **6**(2), 23-30. (Cap. 18)

Czarnecki, K. e Eisenecher, U. (2000). *Generative programming: methods, tools, and applications.* Boston: Addison-Wesley. (Cap. 18)

Davis, A. M. (1993). *Software requirements: objects, functions, & states.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 6)

Dehbonei, B. e Mejia, F. (1995). "Formal development of safety-critical software systems in railway signalling". In *Applications of formal methods* (M. Hinchey e J. P. Bowen, eds.). Londres: Prentice Hall, 227-52. (Cap. 10)

DeMarco, T. (1978). *Structured analysis and system specification.* Nova York: Yourdon Press. (Cap. 8)

DeMarco, T. e Boehm, B. (2002). "The agile methods fray". *IEEE computer*, **35**(6), 90-2. (Cap. 17)

DeMarco, T. e Lister, T. (1985). "Programmer performance and the effects of the workplace". *Proc. 8th int. conf. on software engineering*, Londres: IEEE Press. (Cap. 25)

DeMarco, T. e Lister, T. (1999). *Peopleware: productive projects and teams.* Nova York: Dorset House. (Cap. 25)

DeMarco, T. (1978). *Structured analysis and system specification.* Nova York: Yourdon Press. (Cap. 1)

Diaper, D. (1989). *Task analysis for human-computer interaction.* Chichester: Ellis Horwood. (Cap. 16)

Dijkstra, E. W. (1968). "Cooperating sequential processes". In *Programming languages* (F. Genuys, ed.). Londres: Academic Press, 43-112. (Cap. 15)

Dijkstra, E. W. (1968). "Goto statement considered harmful". *Comm. ACM*, **11**(3), 147-8. (Cap. 20)

Dijkstra, E. W., Dahl, O. J. et al. (1972). *Structured programming.* Londres: Academic Press. (Caps. 23, 32)

Dix, A., Finlay, J. et al. (2004). *Human computer interaction, 3rd edn.* Harlow: Addison-Wesley. (Cap. 16)

Douglass, B. P. (1999). *Real-time UML: developing efficient objects for embedded systems, 2nd edn.* Boston: Addison-Wesley. (Cap. 15)

DuBois, P. (1996). *Software portability with imake, 2nd edn.* Sebastopol, CA: O'Reilly & Associates. (Cap. 29)

Easterbrook, S. e Nuseibeh, B. (1996). Using ViewPoints for inconsistency management. *BCS/IEE software eng. j.*, **11**(1), 31-43.

Easterbrook, S., Lutz, R. et al. (1998). "Experiences using lightweight formal methods for requirements modeling". *IEEE trans. on software engineering*, **24**(1), 4-14. (Cap. 10)

ECMA. (1991). "A reference model for frameworks of computer-assisted software engineering environments". In *Reprints of the seventh international software process workshop*, Yountville, CA: ACM Press. (Cap. 11)

Ehrlich, W., Prasanna, B. et al. (1993). "Determining the cost of a stop-test decision". *IEEE software*, **9**(4), 33-42. (Cap. 24)

El-Amam, K. (2001). *Object-oriented metrics: a review of theory and practice.* (Cap. 27)

Elliott, J., Eckstein, R. et al. (2002). *Java swing, 2nd edn.* Sebastopol, CA: O'Reilly & Associates Inc. (Cap. 16)

Ellison, R. J., Fisher, D. A. et al. (1999). "Survivability: protecting your critical systems". *IEEE internet computing*, **3**(6), 55-63. (Cap. 3)

Ellison, R. J., Linger, R. C. et al. (1999). "Survivable network system analysis: a case study". *IEEE software*, **16**(4), 70-7. (Cap. 3)

Ellison, R. J., Fisher, D. A. e*t al.* (1999a). "Survivability: protecting your critical systems". *IEEE internet computing*, **3**(6), 55-63. (Cap. 30)

Ellison, R., Linger, R. et al. (2002). "Foundations of survivable systems engineering". *Crosstalk: The journal of defense software engineering*, **12**, 10-15. (Caps. 3, 30)

Elrad, T., Askit, M. et al. (2001). "Discussing aspects of AOP". *Comm. ACM*, **44**(10), 33-8. (Cap. 18)

Endres, A. (1975). "An analysis of errors and their causes in system programs". *IEEE trans. on software engineering*, **SE-1**(2), 140-9. (Cap. 3)

Erl, T. (2004). *Service-oriented architecture: a field guide to integrating XML and Web services.* Upple Saddle River, NJ: Prentice-Hall. (Cap. 31)

Erl, T. (2005). *Service-oriented architecture: concepts, technology and design*, Upper Saddle River, NJ: Prentice-Hall. (Cap. 31)

Erlikh, L. (2000). "Leveraging legacy system dollars for e-business". *IT Pro*, março/junho de 2000, 17-23. (Cap. 21)

Estublier, J. e Casallas, R. (1994). "The Adele configuration manager". In *Configuration management* (W. Tichy, ed.). Chichester: John Wiley & Sons, 99-134. (Cap. 29)

Evans, D. e Larochelle, D. (2002). "Improving security using extensible lightweight static analysis". *IEEE software*, **19**(1), 42-51. (Caps. 22, 30)

Ewald, T. (2001). "Overview of COM+". In *Component-Based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 573-88. (Cap. 19)

Fagan, M. E. (1976). "Design and code inspections to reduce errors in program development". *IBM systems j.*, **15**(3), 182-211. (Cap. 22)

Fagan, M. E. (1986). "Advances in software inspections". *IEEE trans. on software engineering*, **SE-12**(7), 744-51. (Cap. 22)

Fayad, M. E. e Schmidt, D. C. (1997). "Object-oriented application frameworks". *Comm. ACM*, **40**(10), 32-8. (Cap. 18)

Feldman, S. I. (1979). "MAKE — a program for maintaining computer programs". *Software-practice and experience*, **9**(4), 255-65. (Cap. 29)

Finkelstein, A., Kramer, J. e*t al.* (1992). "Viewpoints: a framework for integrating multiple perspectives in system development". *Int. j. software engineering knowledge engineering*, **2**(1), 31-58.

Firesmith, D. G. (2003). "Engineering security requirements". *Journal of object technology*, **2**(1), 53-68. (Cap. 9)

Foster, I., Kesselman, C. et al. (2002). "Grid services for distributed system integration". *IEEE computer*, **35**(6), 37-46. (Cap. 12)

Frewin, G. D. e Hatton, B. J. (1986). "Quality management — procedures and practises". *IEE/BCS software engineering j.*, **1**(1), 29-38. (Cap. 22)

Fromme, B. e Walker, J. (1993). "An open architecture for tool and process integration". *Proc. 6th conf. on software engineering environments*, Reading, UK: IEEE Press. (Cap. 11)

Fuggetta, A. (1993). "A classification of CASE technology". *IEEE computer*, **26**(12), 25-38. (Cap. 4)

Fujiwara, E. e Pradhan, D. K. (1990). "Error-control coding in computers". *IEEE computer*, **23**(7), 63-72. (Cap. 20)

Furey, S. e Kitchenham, B. (1997). "Point/counterpoint: function points". *IEEE software*, **14**(2), 28-31. (Cap. 26)

Futatsugi, K., Goguen, J. A. et al. (1985). "Principles of OBJ2". *Proc. 12th ACM symp. on principles of programming languages*, New Orleans: ACM Press. (Cap. 10)

Gamma, E., Helm, R. et al. (1995). *Design patterns: elements of reusable object-oriented software.* Reading, MA: Addison-Wesley. (Cap. 18)

Gane, C. e Sarson, T. (1979). *Structured systems analysis.* Englewood Cliffs, NJ: Prentice Hall. (Caps. 4, 8)

Garlan, D. e Shaw, M. (1993). "An introduction to software architecture". *Advances in software engineering and knowledge engineering*, **1**, 1-39. (Caps. 11, 13)

Garlan, D., Allen, R. et al. (1995). "Architectural mismatch: why reuse is so hard". *IEEE software*, **12**(6), 17-26. (Cap. 18)

Garlan, D., Kaiser, G. E. et al. (1992). "Using tool abstraction to compose systems". *IEEE computer*, **25**(6), 30-8. (Cap. 11)

Garmus, D. e Herron, D. (2000). *Function Point analysis: measurement practices for successful software projects.* Boston: Addison-Wesley. (Cap. 26)

Gilb, T. e Graham, D. (1993). *Software inspection.* Wokingham: Addison-Wesley. (Cap. 22)

Goldberg, A. e Robson, D. (1983). *Smalltalk-80: the language and its implementation.* Reading, MA: Addison-Wesley. (Cap. 16)

Gollmann, D. (1999). *Computer security.* Chichester: John Wiley & Sons. (Cap. 24)

Gomaa, H. (1993). *Software design methods for concurrent and real-time systems.* Reading, MA: Addison-Wesley. (Cap. 15)

Gordon, V. S. e Bieman, J. M. (1995). "Rapid prototyping: lessons learned". *IEEE software*, **12**(1), 85-95. (Cap. 17)

Gotterbarn, D., Miller, K. e*t al.* (1999). "Software engineering code of ethics is approved". *Comm. ACM*, **42**(10), 102-7. (Cap. 1)

Gradecki, J. D. e Lezeiki, N. (2003). *Mastering AspectJ: aspect-oriented programming in Java*, Nova York: John Wiley & Sons.

Grady, R. B. (1993). "Practical results from measuring software quality". *Comm. ACM*, **36**(11), 62-8. (Cap. 27)

Grady, R. B. e Van Slack, T. (1994). "Key lessons in achieving widespread inspection use". *IEEE software*, **11**(4), 46-57. (Cap. 22)

Graham, I. (1994). *Object-oriented methods, 2nd edn.* Wokingham: Addison-Wesley. (Cap. 14)

Griss, M. L. e Wosser, M. (1995). "Making reuse work at Hewlett-Packard". *IEEE software*, **12**(1), 105-7. (Cap. 18)

Groff, J. R., Weinberg, P. N. et al. (2002). "*SQL: the complete reference, 2nd edn*". Nova York: McGraw-Hill Osborne. (Cap. 17)

Grudin, J. (1989). "The case against user interface consistency". *Comm. ACM*, **32**(10), 1164-73. (Cap. 16)

Guimaraes, T. (1983). "Managing application program maintenance expenditures". *Comm. ACM*, **26**(10), 739-46. (Cap. 21)

Gunning, R. (1962). *Techniques of clear writing.* Nova York: McGraw-Hill. (Cap. 27)

Guttag, J. (1977). "Abstract data types and the development of data structures". *Comm. ACM*, **20**(6), 396-405. (Cap. 10)

Guttag, J., Horning, J. et al. (1993). *Larch: languages and tools for formal specification.* Heidelberg: Springer-Verlag. (Cap. 10)

Haase, V., Messnarz, R. et al. (1994). "Bootstrap: fine tuning process assessment". *IEEE software*, **11**(4), 25-35. (Cap. 28)

Hall, A. (1990). "Seven myths of formal methods". *IEEE software*, **7**(5), 11-20. (Cap. 10)

Hall, A. (1996). "Using formal methods to develop an ATC information system". *IEEE software*, **13**(2), 66-76. (Caps. 3, 9, 10)

Hall, A. e Chapman, R. (2002). "Correctness by construction: developing a commercially secure system". *IEEE software*, **19**(1), 18-25. (Caps. 3, 9, 10)

Hall, E. (1998). *Managing risk: methods for software systems development.* Reading, MA: Addison-Wesley. (Cap. 5)

Hall, T. e Fenton, N. (1997). "Implementing effective software metrics programs". *IEEE software*, **14**(2), 55-64. (Cap. 27)

Halstead, M. H. (1977). *Elements of software science.* Amsterdam: North-Holland. (Cap. 21)

Hamlet, D. (1992). "Are we testing for true reliability?" *IEEE software*, **9**(4), 21-7. (Cap. 24)

Hammer, M. (1990). "Reengineering work: don't automate, obliterate". *Harvard business review*, julho/agosto de 1990, 104-12. (Cap. 28)

Hammer, M. e McLeod, D. (1981). "Database descriptions with SDM: a semantic database model". *ACM trans. on database sys.*, **6**(3), 351-86. (Cap. 8)

Hardin, D., Frerking, M. et al. (2002). "Getting down and dirty: device-level programming using the real-time specification for Java". *Proc. fifth IEEE international symp. on object-oriented real-time distributed computing*. Washington, DC: IEEE Computer Society Press. (Cap. 15)

Harel, D. (1987). "Statecharts: a visual formalism for complex systems". *Sci. comput. programming*, **8**(3), 231-74. (Caps. 8, 14, 15)

Harel, D. (1988). "On visual formalisms". *Comm. ACM*, **31**(5), 514-30. (Caps. 8, 15)

Harold, E. R. e Means, W. S. (2002). *XML in a nutshell.* Sebastopol. CA: O'Reilly & Associates. (Cap. 13)

Hass, A. M. J. (2003). *Configuration management: principles and practice.* Boston: Addison-Wesley. (Cap. 29)

Hayes, I. (1987). *Specification case studies.* Londres: Prentice Hall. (Cap. 10)

Heninger, K. L. (1980). "Specifying software requirements for complex systems: new techniques and their applications". *IEEE trans. on software engineering*, **SE-6**(1), 2-13. (Cap. 6)

Highsmith, J. A. (2000). *Adaptive software development: a collaborative approach to managing complex systems.* Nova York: Dorset House. (Cap. 17)

Higuera-Toledano, M. T. e Issarny, V. (2000). "Java embedded real-time systems: an overview of existing solutions". *Proc. third IEEE international symp. on object-oriented real-time distributed computing*. Newport Beach, CA: IEEE Computer Society Press. (Cap. 15)

Hoare, C. A. R. (1974). "Monitors: an operating system structuring concept". *Comm. ACM*, **21**(8), 666-77. (Cap. 15)

Hoare, C. A. R. (1985). *Communicating sequential processes.* Londres: Prentice Hall. (Cap. 10)

Hofmeister, C., Nord, R. et al. (2000). *Applied software architecture.* Boston: Addison-Wesley. (Cap. 11)

Horswill, J. e Miller, S. A. (2000). *Designing and programming CICS applications.* Sebastopol, CA: O'Reilly & Associates. (Cap. 13)

Huang, Y. e Kintala, C. M. R. (1993). "Software implemented fault tolerance: technologies and experience". *Proc. 23rd fault-tolerant computing symposium (FTCS-23)*, Toulouse, França: IEEE Computer Society Press. (Cap. 20)

Huff, C. C. (1992). "Elements of a realistic CASE tool adoption budget". *Comm. ACM*, **35**(4), 45-54. (Cap. 4)

Huff, K. E. (1996). "Software process modeling". In *Trends in software: software process* (A. Fuggetta e A. Wolf, eds.). Chichester: John Wiley & Sons, 1-24. (Cap. 28)

Huff, C. e Martin, C. D. (1995). "Computing consequences: a framework for teaching ethical computing". *Comm. ACM*, **38**(12), 75-84. (Cap. 1)

Hughes, J. A., O'Brien, J. et al. (1997). "Designing with ethnography: a presentation framework for design". *Proc. DIS'97*, Amsterdam: ACM Press. (Cap. 16)

Hull, R. e King, R. (1987). "Semantic database modeling: survey, applications and research issues". *ACM computing surveys*, **19**(3), 201-60. (Cap. 8)

Humphrey, W. (1989). *Managing the software process.* Reading, MA: Addison-Wesley. (Caps. 22, 27. 28)

Humphrey, W. S. (1988). "Characterizing the software process". *IEEE software*, **5**(2), 73-9. (Cap. 28)

Humphrey, W. S. (1995). *A discipline for software engineering.* Reading, MA: Addison-Wesley. (Cap. 28)

IEC. (1998). *Standard IEC 61508: functional safety of electrical/electronic/programmable electronic safety-related systems*. (Cap. 9)

IEEE. (1998). "IEEE recommended practice for software requirements specifications". In *IEEE software engineering standards collection*. Los Alamitos, CA: IEEE Computer Society Press. (Cap. 6)

IEEE. (2003). *IEEE software engineering standards collection on CD-ROM.* Los Alamitos, CA: IEEE Computer Society Press. (Cap. 27)

Ince, D. (1994). *ISO 9001 and software quality assurance.* Londres: McGraw-Hill. (Cap. 27)

Jackson, M. A. (1983). *System development.* Londres: Prentice Hall. (Caps. 1, 8)

Jackson, M. A. (1995). *Requirements and specifications.* Wokingham: Addison-Wesley. (Cap. 6)

Jacky, J. (1995). Specifying a safety-critical control system. *IEEE trans. on software engineering*, **21**(2), 99-106. (Cap. 10)

Jacky, J. (1997). *The way of Z: practical programming with formal methods.* Cambridge, UK: Cambridge University Press. (Cap. 10)

Jacky, J., Unger, J. et al. (1997). "Experience with Z: developing a control program for a radiation therapy machine". *Proc. ZUM'97*, Reading: Springer. (Cap. 10)

Jacobsen, I. e Ng, P.-W. (2004). *Aspect-oriented software development with use cases*. Boston, MA: Addison Wesley.

Jacobsen, I., Christerson, M. et al. (1993). *Object-oriented software engineering.* Wokingham: Addison-Wesley. (Caps. 6, 8, 15)

Jacobsen, I., Griss, M. et al. (1997). *Software reuse.* Reading, MA: Addison-Wesley. (Caps. 18, 19)

Jahanian, F. e Mok, A. K. (1986). "Safety analysis of timing properties in real-time systems". *IEEE trans. on software engineering*, **SE-12**(9), 890-904. (Cap. 9)

Janis, I. L. (1972). *Victims of groupthink: a psychological study of foreign policy decisions and fiascos.* Boston: Houghton Mifflin. (Cap. 25)

Jelinski, Z. e Moranda, P. B. (1972). "Software reliability research". In *Statistical computer performance evaluation* (W. Frieberger, ed.). Nova York: Academic Press, 465-84. (Cap. 24)

Johnson, P. L. (1993). *ISO 9000: meeting the new international standards.* Nova York: McGraw-Hill. (Cap. 27)

Jones, C. B. (1980). *Software development — a rigorous approach.* Londres: Prentice Hall. (Cap. 10)

Jones, C. B. (1986). *Systematic software development using VDM.* Londres: Prentice Hall. (Caps. 10, 22)

Kafura, D. e Reddy, G. R. (1987). "The use of software complexity metrics in software maintenance". *IEEE trans. on software engineering*, **SE-13**(3), 335-43. (Cap. 21)

Kan, S. H. (2003). *Metrics and models in software quality engineering.* Boston: Addison-Wesley. (Cap. 24)

Katz, S. (2005). "A survey of verification and static analysis for aspects". http://www.aosd-europe.net/ documents/verificM81.pdf.

Kavantzas, N., Burdett, D. e*t al.* (2004). "Web Services choreography description language version 1.0". http://www.w3.org/TR/2004/ WD-ws-cdl-10-20040427/. (Cap. 31)

Kiczales, G., Hilsdale, E. et al. (2001). "Getting started with AspectJ". *Comm. ACM*, **44**(10), 59-65. (Caps. 18, 32)

Kilpi, T. (2001). "Implementing a software metrics program at Nokia". *IEEE software*, **18**(6), 72-7. (Cap. 27)

Kit, E. (1995). *Software testing in the real world: improving the process.* Reading, MA: Addison-Wesley. (Cap. 22)

Kitchenham, B. (1990). "Measuring software development". In *Software reliability handbook* (P. Rook, ed.). Amsterdam: Elsevier, 303-31. (Cap. 27)

Kleppe, A., Warmer, J. et al. (2003). *MDA explained: the model-driven architecture — practice and promise.* Boston: Addison-Wesley. (Cap. 14)

Knight, J. C. e Leveson, N. G. (1986). "An experimental evaluation of the assumption of independence in multi-version programming". *IEEE trans. on software engineering*, **SE-12**(1), 96-109. (Cap. 20)

Knight, J. C. e Leveson, N. G. (2002). "Should software engineers be licensed?" *Comm. ACM*, **45**(11), 87-90. (Cap. 24)

Knuth, D. E. (1971). *The art of computer programming: fundamental algorithms.* Reading, MA: Addison-Wesley. (Cap. 18)

Kotonya, G. e Sommerville, I. (1996). "Requirements engineering with viewpoints". *BCS/IEE Software eng. j.*, **11** (1), 5-18.

Kotonya, G. e Sommerville, I. (1998). *Requirements engineering: processes and techniques.* Chichester: John Wiley & Sons. (Cap. 6)

Kreger, H. (2001). *Web services conceptual architecture (WSCA 1.0).* IBM. www.ibm.com/software/solutions/ webservices/pdf/WSCA. pdf (Cap. 12)

Krutchen, P. (2000). *The rational unified process — an introduction.* Reading, MA: Addison-Wesley. (Caps. 4, 8)

Kumaran, S. I. (2001). *JINI technology: an overview.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 12)

Kuvaja, P., Similä, J. et al. (1994). *Software process assessment and improvement: the Bootstrap approach.* Oxford: Blackwell Publishers. (Cap. 28)

Laddad, R. (2003a). *AspectJ in action*, Greenwich, CT: Manning Publications Co.

Laddad, R. (2003b). *AspectJ in action: practical aspect-oriented programming*, Greenwich, CT: Manning Publications.

Lamping, J., Rao, R. et al. (1995). "A focus + context technique based on hyperbolic geometry for visualising large hierarchies". *Proc. CHI'95*, Denver, CO: ACM Press. (Cap. 16)

Laprie, J.-C. (1995). "Dependable computing: concepts, limits, challenges". *Proc. 25th IEEE symposium on fault-tolerant computing*, Pasadena, CA: IEEE Press. (Cap. 3)

Laprie, J.-C., Arlat, J. et al. (1995). "Architectural issues in software fault tolerance". In *Software fault tolerance* (M. R. Lyu, ed.). Nova York: John Wiley & Sons 47-80. (Cap. 20)

Larman, C. (2002). *Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 17)

Larus, J. R., Ball, T. e*t al.* (2004). "Righting software". *IEEE software*, **21**(3), 92-100. (Cap. 30)

Laudon, K. (1995). "Ethical concepts and information technology". *Comm. ACM*, **38**(12), 33-9. (Cap. 1)

Leblang, D. B. e Chase, R. P. (1987). "Parallel software configuration management in a network environment". *IEEE software*, **4**(6), 28-35. (Cap. 29)

Lehman, M. M. (1996). "Laws of software evolution revisited". *Proc. european workshop on software process technology (EWSPT'96)*, Nancy, França: Springer-Verlag. (Cap. 21)

Lehman, M. M. e Belady, L. (1985). *Program evolution: processes of software change.* Londres: Academic Press. (Cap. 21)

Lehman, M. M., Perry, D. E. et al. (1998). "On evidence supporting the FEAST hypothesis and the laws of software evolution". *Proc. Metrics'98*, Bethesda, MD: IEEE Computer Society Press. (Cap. 21)

Lehman, M. M., Ramil, J. F. et al. (2001). "An approach to modelling long-term growth trends in software systems". *Proc. int. conf. on software maintenance*, Florence, Italy: IEEE Computer Society Press. (Cap. 21)

Leveson, N. e Stolzy, J. (1987). "Safety analysis using Petri nets". *IEEE transactions on software engineering*, **13**(3), 386-97. (Cap. 9)

Leveson, N. G. (1985). Software safety. In *Resilient computing systems* (T. Anderson, ed.). Londres: Collins, 12343. (Caps. 3, 9)

Leveson, N. G. (1995). *Safeware: system safety and computers.* Reading, MA: Addison-Wesley. (Caps. 9, 20)

Leveson, N. G. e Harvey, P. R. (1983). Analysing software safety. *IEEE trans. on software engineering*, **SE-9**(5), 569-79. (Cap. 9)

Lewis, P. M., Bernstein, A. J. et al. (2003). *Databases and transaction processing: an application-oriented approach.* Boston: Addison-Wesley. (Cap. 13)

Lientz, B. P. e Swanson, E. B. (1980). *Software maintenance management.* Reading, MA: Addison-Wesley. (Cap. 21)

Linger, R. C. (1994). "Cleanroom process model". *IEEE software*, **11**(2), 50-8. (Caps. 4, 22)

Liskov, B. e Guttag, J. (1986). *Abstraction and specification in program development.* Cambridge, MA: MIT Press. (Cap. 10)

Littlewood, B. (1990). "Software reliability growth models". In *Software reliability handbook* (P. Rook, ed.). Amsterdam: Elsevier, 401-12. (Caps. 3, 24)

Littlewood, B. e Verrall, J. L. (1973). "A Bayesian reliability growth model for computer software". *Applied statistics*, **22**, 332-46. (Cap. 24)

Livshits, V. B. e Lam, M. S. (2005). "Finding security vulnerabilities in java applications with static analysis". *Proc. 14th USENIX security symposium*, Baltimore, MD. http://www.usenix.org/events/sec05/tech/.

Londeix, B. (1987). *Cost estimation for software development.* Wokingham: Addison-Wesley. (Cap. 26)

Lovelock, C., Vandermerwe, S. et al. (1996). *Services marketing.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 12)

Lutz, M. (1996). *Programming python.* Sebastopol, CA: O'Reilly & Associates. (Cap. 17)

Lutz, R. R. (1993). "Analysing software requirements errors in safety-critical embedded systems". *Proc. RE'93*, San Diego CA: IEEE Computer Society Press. (Caps. 3, 22, 23)

MacDonell, S. G. (1994). "Comparative review of functional complexity assessment methods for effort estimation". *BCS/IEE software engineering J.*, **9**(3), 107-17. (Cap. 26)

Marshall, J. E. e Heslin, R. (1975). "Boys and girls together: sexual composition and the effect of density on group size and cohesiveness". *J. of personality and social psychology*, **35**(5), 952-61. (Cap. 25)

Martin, D., Rodden, T. et al. (2001). "Finding patterns in the fieldwork". *Proc. ECSCW'01*, Bonn: Kluwer. (Cap. 18)

Martin, D., Rouncefield, M. et al. (2002). "Applying patterns of interaction to work (re)design: e-government and planning". *Proc CHI'2002*, ACM Press. (Cap. 18)

Maslow, A. A. (1954). *Motivation and personality.* Nova York: Harper and Row. (Cap. 25)

Massol, V. and Husted, T. (2003). *JUnit in Action.* Greenwich, CT: Manning. (Cap. 23)

Matsumoto, Y. (1984). "Some experience in promoting reusable software: presentation in higher abstract levels". *IEEE trans. on software engineering*, **SE-10**(5), 502-12. (Cap. 18)

McCabe, T. J. (1976). "A complexity measure". *IEEE trans. on software engineering*, **SE-2**(4), 308-20. (Cap. 21)

McCue, G. M. (1978). "IBM's Santa Teresa laboratory: architectural design for program development". *IBM systems j.*, **17**(1), 4-25. (Cap. 25)

McDougall, P. (2000). "The power of peer-to-peer". *Information week*, August 28, http://www.informationweek.com. (Cap. 12)

McGuffin, R. W., Elliston, A. E. et al. (1979). "CADES — software engineering in practice". *Proc. 4th int. conf. on software engineering*, Munich: IEEE Computer Society Press. (Cap. 11)

McIlroy, M. D. (1968). "Mass-produced software components". *Proc. NATO conf. on software engineering*, Garmisch, Germany: Springer-Verlag. (Cap. 18)

Meyer, B. (1992). "Design by contract". *IEEE computer*, **25**(10), 40-51. (Cap. 19)

Meyer, B. (2003). "The grand challenge of trusted components". *Proc. ICSE 25: int. conf. on software engineering*, Portland, OR: IEEE Press. (Cap. 19)

Mili, H., Mili, A. et al. (2002). *Reuse-based software engineering.* Nova York: John Wiley & Sons. (Cap. 19)

Miller, G. A. (1957). "The magical number 7 plus or minus two: some limits on our capacity for processing information". *Psychological review*, **63**, 81-97. (Cap. 16)

Mills, H. D., Dyer, M. et al. (1987). "Cleanroom software engineering". *IEEE software*, **4**(5), 19-25. (Caps. 3, 4, 22)

Mitschele-Thiel, A. (2001). *Systems engineering with sdl: developing performance-critical communication systems.* Chichester: John Wiley & Sons. (Cap. 22)

MOD, (1995). *The procurement of safety critical software (revised edn).* UK Ministry of Defence, Interim Standard, 00-55. (Cap. 22)

Mosley, D. J. e Posey, B. A. (2002). *Just enough test automation.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 23)

Mumford, E. (1989). "User participation in a changing environment — why we need it". In *Participation in systems development* (K. Knight, ed.). Londres: Kegan Paul. (Cap. 2)

Munch, B. P., Larsen, J-O. et al. (1993). "Uniform versioning: the change-oriented model". *Proc. 4th workshop on software configuration management*, Baltimore, MD: ACM Press. (Cap. 29)

Musa, J. D. (1993). "Operational profiles in software reliability engineering". *IEEE software*, **10**(2), 14-32. (Cap. 24)

Musa, J. D. (1998). *Software reliability engineering: more reliable software, faster development and testing.* Nova York: McGraw-Hill. (Cap. 24)

Musciano, C. e Kennedy, B. (2002). *HTML & XHTML: the definitive guide.* Sebastopol, CA: O'Reilly & Associates. (Cap. 16)

Myers, W. (1989). "Allow plenty of time for large-scale software". *IEEE software*, **6**(4), 92-9. (Cap. 26)

Nakajo, T. e Kume, H. (1991). "A case history analysis of software error-cause relationships". *IEEE trans. on software engineering*, **18**(8), 830-8. (Cap. 3)

Neil, M., Ostrolenk, G. et al. (1998). "Lessons from using Z to specify a software tool". *IEEE trans. on software engineering*, **24**(1), 15-23. (Cap. 10)

Neilsen, J. (1993). *Usability engineering. Nova York: Academic Press. (Cap. 16)*

Newcomer, E. & Lomow, G. (2005). *Understanding SOA with Web services*. Boston, MA: Addison Wesley. (Cap. 31)

Nii, H. P. (1986). "Blackboard systems, parts 1 and 2". *AI magazine*, **7**(3 and 4), 38-53 and 62-9. (Cap. 11)

Nilsen, K. (1998). "Adding real-time capabilities to Java". *Comm. ACM*, **41**(6), 49-56. (Cap. 15)

Norman, D. A. e Draper, S. W. (1986). *User-centered system design.* Hillsdale, NJ: Lawrence Erlbaum. (Cap. 16)

Nosek, J. T. e Palvia, P. (1990). "Software maintenance management: changes in the last decade". *Software maintenance: research and practice*, **2**(3), 157-74. (Cap. 21)

Nuseibeh, B. (1997). "Ariane 5: who dunnit?" *IEEE software*, **14**(3), 15-16. (Cap. 18)

O'Connor, J., Mansour, C. et al. (1994). "Reuse in command and control systems". *IEEE software*, **11**(4), 70-9. (Cap. 18)

Offen, R. J. e Jeffrey, R. (1997). "Establishing software measurement programs". *IEEE software*, **14**(2), 45-54. (Cap. 27)

O'Leary, D. E. (2000). *Enterprise resource planning systems: systems, life cycle, electronic commerce and risk.* Cambridge, UK: Cambridge University Press. (Cap. 18)

Oram, A. (2001). *Peer-to-peer: harnessing the power of disruptive technologies*. Sebastopol, CA: O'Reilly & Associates. (Cap. 12)

Oram, A. e Talbott, S. (1991). *Managing projects with make, 2nd edn.* Sebastopol, CA: O'Reilly & Associates. (Cap. 29)

Orcero, D. S. (2000). "The code analyser LCLint". *Linux journal*, 73, http://www.linuxjournal.com/article.php?sid=3599. (Cap. 22)

Orfali, R. e Harkey, D. (1998). *Client/server programming with Java and CORBA.* Nova York: John Wiley & Sons. (Cap. 12)

Oskarsson, O. e Glass, R. L. (1995). *An ISO 9000 approach to building quality software.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 27)

Ould, M. A. (1999). *Managing software quality and business risk.* Chichester: John Wiley & Sons. (Cap. 5)

Ould, M. A. (1995). *Business processes: modelling and analysis for re-engineering and improvement.* Chichester: John Wiley & Sons. (Cap. 28)

Ousterhout, J. (1994). *TCL and the TK toolkit.* Reading, MA: Addison-Wesley. (Cap. 17)

Ousterhout, J. K. (1998). "Scripting: higher-level programming for the 21st century". *IEEE computer*, **31**(3), 23-30. (Caps. 17, 26)

Owl_Services_Coalition (2003). "OWL-S: semantic markup for Web services". http://www.daml.org/services/owl-s/1.0/owl-s.pdf. (Cap. 31)

Palmer, S. R. e Felsing, J. M. (2002). *A practical guide to feature-driven development.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 17)

Parnas, D. L., van Schouwen, J. et al. (1990). "Evaluation of safety-critical software". *Comm. ACM*, **33**(6), 636-51. (Caps. 20, 24)

Paulk, M. C. e Konrad, M. (1994). "An overview of ISO's SPICE project". *IEEE computer*, **27**(4), 68-70. (Cap. 28)

Paulk, M. C., Curtis, B. et al. (1993). "Capability maturity model, version 1.1". *IEEE software*, **10**(4), 18-27. (Cap. 28)

Paulk, M. C., Weber, C. V. et al. (1995). *The capability maturity model: guidelines for improving the software process.* Reading, MA: Addison-Wesley. (Caps. 28, 29)

Peach, R. W. (1996). *The ISO 9000 handbook, 3rd edn.* Nova York: Irwin Professional. (Caps. 27, 29)

Perrow, C. (1984). *Normal accidents: living with high-risk technology.* Nova York: Basic Books. (Cap. 3)

Peterson, J. L. (1981). *Petri net theory and the modeling of systems.* Nova York: McGraw-Hill. (Caps. 9, 10)

Pfaff, G. e ten Hagen, P. J. W. (1985). *Seeheim workshop on user interface management systems.* Heidelberg: Springer-Verlag. (Cap. 16)

Pfarr, T. e Reis, J. E. (2002). "The integration of COTS/GOTS within NASA's HST command and control system". *Proc. ICCBSS 2002 (1st int. conf on COTS-based software systems)*, Orlando, FL: Springer-Verlag. (Cap. 18)

Pfleeger, C. P. (1997). *Security in computing, 2nd edn.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 3)

Pfleeger, C. P. e Pfleeger, S. L. (2003). *Security in computing*, 3ª ed., Upper Saddle River, NJ: Prentice-Hall. (Cap. 30)

Pope, A. (1998). *CORBA.* Harlow: Addison-Wesley. (Cap. 19)

Potter, B., Sinclair, J. et al. (1996). *An introduction to formal specification and Z.* Londres: Prentice Hall. (Cap. 10)

Preiser, W. e Ostoff, E. (2001). *The universal design handbook.* Nova York: McGraw-Hill. (Cap. 16)

Pritchard, J. (1999). *COM and CORBA side by side: architectures, strategies, and implementations.* Boston: Addison-Wesley. (Cap. 12)

Prowell, S. J., Trammell, C. J. et al. (1999). *Cleanroom software engineering: technology and process.* Reading, MA: Addison-Wesley. (Caps. 4, 10, 22, 24)

Pulford, K., Kuntzmann-Combelles, A. et al. (1996). *A quantitative approach to software management.* Wokingham: Addison-Wesley. (Caps. 27, 28)

Pullum, L. L. (2001). *Software fault tolerance techniques and implementation.* Norwood, MA: Artech House. (Cap. 20)

Putnam, L. H. (1978). "A general empirical solution to the macro software sizing and estimating problem". *IEEE trans. on software engineering*, **SE-4**(3), 345-61. (Cap. 26)

Randell, B. (1975). "System structure for software fault tolerance". *IEEE trans. on software engineering*, **SE-1**(2), 220-32. (Cap. 20)

Randell, B. e Xu, J. (1995). "The evolution of the recovery block concept". In *Software fault tolerance* (M. R. Lyu, ed.). Nova York: John Wiley & Sons, 1-22. (Cap. 20)

Rankin, C. (2002). "The software testing automation framework". *IBM systems j.*, **41**(1), 126-40. (Cap. 23)

Redmill, F. (1998). "IEC 61508: principles and use in the management of safety". *IEEE computing and control engineering j.*, **9**(10), 205-13. (Cap. 9)

Reiss, S., P. (1990). "Connecting tools using message passing in the field environment". *IEEE software*, **7**(4), 57-66. (Cap. 11)

Rettig, M. (1994). "Practical programmer: prototyping for tiny fingers". *Comm. ACM*, **37**(4), 21-7. (Cap. 17)

Rittel, H. e Webber, M. (1973). "Dilemmas in a general theory of planning". *Policy sciences*, 4, 155-69. (Cap. 2)

Robertson, S. e Robertson, J. (1999). *Mastering the requirements process.* Harlow: Addison-Wesley. (Cap. 6)

Robinson, P. J. (1992). *Hierarchical object-oriented design.* Englewood Cliffs, NJ: Prentice Hall. (Caps. 4, 8, 14)

Ross, D. T. (1977). "Structured analysis (SA): a language for communicating ideas". *IEEE trans. on software engineering*, **SE-3**(1), 16-34. (Cap. 6)

Royce, W. W. (1970). "Managing the development of large software systems: concepts and techniques". *Proc. IEEE WESTCON*, Los Angeles CA: IEEE Computer Society Press. (Cap. 4)

Rubin, K. e Goldberg, A. (1992). "Object behaviour analysis". *Comm. ACM*, **35**(9), 48-62. (Cap. 14)

Rumbaugh, J., Blaha, M. et al. (1991). *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice Hall. (Caps. 1, 4, 8)

Rumbaugh, J., Jacobson, I. et al. (1999). *The unified modeling language reference manual.* Reading, MA: Addison-Wesley. (Caps. 1, 4, 8, 14)

Rumbaugh, J., Jacobson, I. et al. (1999). *The unified software development process.* Reading, MA: Addison-Wesley. (Caps. 1, 4, 8)

Sackman, H., Erikson, W. J. et al. (1968). "Exploratory experimentation studies comparing on-line and off-line programming performance". *Comm. ACM*, **11**(1), 3-11. (Cap. 26)

Schmidt, D. C. (1997). "Applying design patterns and frameworks to develop object-oriented communications software". In *Handbook of programming languages*, vol. 1 (P. Salus, ed.). Londres: Macmillan Computer Publishing. (Cap. 18)

Schneidewind, N. F. e Keller, T. W. (1992). Applying reliability models to the space shuttle. *IEEE software*, **9**(4), 28-33. (Cap. 24)

Schneier, B. (2000). *Secrets and lies: digital security in a networked world*, Nova York: John Wiley and Sons.

Schoman, K. e Ross, D. T. (1977). "Structured analysis for requirements definition". *IEEE trans. on software engineering*, **SE-3**(1), 6-15. (Cap. 6)

Schwaber, K. e Beedle, M. (2001). *Agile software development with scrum.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 17)

Scott, J. E. (1999). "The FoxMeyer Drug's bankruptcy: was it a failure of ERP?" *Proc. association for information systems 5th Americas conf. on information systems*, Milwaukee, WI. (Cap. 18)

Selby, R. W., Basili, V. R. et al. (1987). "Cleanroom software development: an empirical evaluation". *IEEE trans. on software engineering*, **SE-13**(9), 1027-37. (Caps. 4, 22)

Sheldon, F. T., Kavi, K. M. et al. (1992). "Reliability measurement: from theory to practice". *IEEE software*, **9**(4), 13-20. (Cap. 24)

Shlaer, S. e Mellor, S. (1988). *Object-oriented systems analysis: modeling the world in data.* Englewood Cliffs, NJ: Yourdon Press. (Cap. 14)

Shneiderman, B. (1998). *Designing the user interface, 3rd edn.* Reading, MA: Addison-Wesley. (Cap. 16)

Siegal, J. (1998). "OMG overview: CORBA and the OMA in enterprise computing". *Comm. ACM*, **41**(10), 37-43. (Cap. 12)

Silberschatz, A., Galvin, P. B. et al. (2002). *Operating system concepts, 6th edn.* Nova York: John Wiley & Sons. (Cap. 15)

Skonnard, A. e Gudgin, M. (2002). *Essential XML quick reference: a programmer's reference to XML, XPath, XSLT, XML Schema, SOAP, and more.* Boston: Addison-Wesley. (Cap. 12)

Snyder, C. (2003). *Paper prototyping: the fast and easy way to design and refine user interfaces.* San Francisco: Morgan Kaufmann. (Cap. 16)

Sommerville, I. e Sawyer, P. (1997). "Viewpoints: principles, problems and a practical approach to requirements engineering". *Annals of software engineering*, **3** 101-30.

Sommerville, I., Sawyer, P. e*t al.* (1998). "Viewpoints for requirements elicitation: a practical approach". *Proc. int. conf. on requirements engineering*, Colorado.

Spafford, E. (1989). "The Internet worm: crisis and aftermath". *Comm. ACM*, **32**(6), 678-87. (Caps. 3, 30)

Spivey, J. M. (1990). "Specifying a real-time kernel". *IEEE software*, **7**(5), 21-8. (Cap. 10)

Spivey, J. M. (1992). *The Z notation: a reference manual, 2nd edn.* Londres: Prentice Hall. (Caps. 10, 22)

Stal, M. (2002). "Web services: beyond component-based computing". *Comm. ACM*, **45**(10), 71-6. (Cap. 12)

Stapleton, J. (1997). *DSDM dynamic systems development method.* Harlow: Addison-Wesley. (Cap. 17)

Stephens, M. e Rosenberg, D. (2003). *Extreme programming refactored.* Berkley, CA: Apress. (Cap. 17)

Stevens, P. e Pooley, R. (1999). *Software engineering with objects and components.* Harlow: Addison-Wesley. (Cap. 6)

Storey, N. (1996). *Safety-critical computer systems.* Harlow: Addison-Wesley. (Caps. 9, 20)

Suchman, L. (1983). "Office procedures as practical action". *ACM trans. on office information systems*, **1**(3), 320-28. (Cap. 16)

Swartz, A. J. (1996). "Airport 95: automated baggage system?" *ACM software engineering notes*, **21**(2), 79-83. (Cap. 2)

Symons, C. R. (1988). "Function-point analysis: difficulties and improvements". *IEEE trans. on software engineering*, **14**(1), 2-11. (Cap. 26)

Szyperski, C. (2002). *Component software: beyond object-oriented programming, 2nd edn.* Harlow: Addison-Wesley. (Caps. 12, 19)

Tanenbaum, A. S. (2001). *Modern operating systems, 2nd edn.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 15)

Thayer, R. H. (1997). "Software system engineering: an engineering process". In *Software requirements engineering* (R. H. Thayer e M. Dorfmann, eds.). Los Alamitos: IEEE Computer Society Press, 84106. (Cap. 2)

Thayer, R. H. (2002). "Software system engineering: a tutorial". *IEEE computer*, **35**(4), 68-73. (Cap. 2)

Tichy, W. (1985). "RCS — a system for version control". *Software practice and experience*, **15**(7), 637-54. (Cap. 29)

Tracz, W. (2001). "COTS myths and other lessons learned in component-based software development". In *Component-based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 99-112. (Cap. 18)

Turner, M., Budgen, D. et al. (2003). "Turning software into a service". *IEEE computer*, **36**(10), 38-45. (Cap. 12)

Ulrich, W. M. (1990). "The evolutionary growth of software reengineering and the decade ahead". *American programmer*, **3**(10), 14-20. (Cap. 21)

van der Aalst, W. M. P. e ter Hofstede, A. H. M. (2005). "YAWL: yet another workflow language". *Information systems*, **30**(4), 245-75.

Vesperman, J. (2003). *Essential CVS.* Sebastopol, CA: O'Reilly & Associates. (Cap. 29)

Viega, J. e Mcgraw, G. (2002). *Building secure software*, Boston, MA: Addison-Wesley. (Cap. 30)

Wall, L., Christiansen, T. et al. (1996). *Programming Perl.* Sebastopol, CA: O'Reilly & Associates. (Cap. 17)

Wang, N., Schmidt, D. C. et al. (2001). "Overview of the CORBA component model". In *Component-based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 557-72. (Cap. 19)

Ward, P. e Mellor, S. (1985). *Structured development for real-time systems.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 8)

Warmer, J. e Kleppe, A. (1998). *The object constraint language: precise modeling with UML.* Boston: Addison-Wesley. (Cap. 19)

Warren, I. (1998). *The renaissance of legacy systems.* Londres: Springer. (Cap. 21)

Weinberg, G. (1971). *The psychology of computer programming.* Nova York: Van Nostrand. (Caps. 17, 25)

Weinreich, R. e Sametinger, J. (2001). "Component models and component services: concepts and principles". In *Component-based software engineering* (G. T. Heineman e W. T. Councill, eds.). Boston: Addison-Wesley, 33-48. (Cap. 19)

Weiss, S. (2002). *Handheld usability.* Nova York: John Wiley & Sons. (Cap. 16)

Westmark, V. R. (2004). "A definition for information system survivability". *Proc. 37th Hawaii int. conf. on system sciences*, Hawaii. (Cap. 30)

Wheeler, D. A. (2003). *Secure programming for Linux and UNix HOWTO*, Web published, http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html. (Cap. 30)

White, B. A. (2000). *Software configuration management strategies and rational ClearCase.* Reading, MA: Addison-Wesley. (Cap. 29)

White, S. A. (2004). "An introduction to BPMN". http://www.bpmn.org/Documents/ Introduction%20to%20BPMN. (Cap. 31)

White, S., Alford, M. e*t al*. (1993). "Systems engineering of computer-based systems". *IEEE computer*, **26**(11), 54-65. (Cap. 2)

Whitgift, D. (1991). *Software configuration management: methods and tools.* Chichester: John Wiley & Sons. (Cap. 29)

Whittaker, J. A. e Thompson, H. H. (2004). *How to break sofware security: effective techniques for security testing*, Boston, MA: Addison-Wesley.

Whittaker, J. W. (2002). *How to break software: a practical guide to testing.* Boston: Addison-Wesley. (Cap. 23)

Williams, L., Kessler, R. R. et al. (2000). "Strengthening the case for pair programming". *IEEE software*, **17**(4), 19-25. (Cap. 17)

Wirfs-Brock, R. J. e Johnson, R. E. (1990). "Surveying current research in object-oriented design". *Comm. ACM*, **33**(9), 104-24. (Cap. 18)

Wirfs-Brock, R., Wilkerson, B. et al. (1990). *Designing object-oriented software.* Englewood Cliffs, NJ: Prentice Hall. (Cap. 14)

Wordsworth, J. (1996). *Software engineering with B.* Wokingham: Addison-Wesley. (Caps. 4, 9, 10, 22)

Wordsworth, J. B. (1991). "The CICS application programming interface definition". *Proc. Z user workshop*. Oxford, Berlin: Springer-Verlag. (Cap. 10)

Zimmermann, H. (1980). "OSI reference model — the ISO model of architecture for open systems interconnection". *IEEE transactions on communications*, **COM-28**(4), 425-32. (Cap. 11)

ÍNDICE REMISSIVO

A

tipo abstratos de dados, 149
abstração, 113
risco aceitável, 131
teste de aceitação, 53
ACM, 10-12
objetos ativos, 212
controles Active X, 182
atividades
    diagrama de barras, 68
    gerenciamento de projeto, 62-63
    processos de software, 49-54
modelos de atividade, 6
redes de atividades, 66-69. *Veja também* gerenciamento de projetos
adaptabilidade, 107
manutenção para adaptar, 334
composição aditiva, 300
ADL (*Architectural Description Language*), 164
recomendação, 513
agregação, 123
métodos ágeis, 262-263
ALARP (*As low as reasonably practical*), 131
especificação algébrica, 149-152
algoritmos
    projeto, 51
    erro de, 134
alias, 318
teste alpha, 53
análise
    de componentes, 46
    modelos de contexto, 113-114
    impacto, 337
    orientada a objetos, 121
    requisitos, 50
    processo de engenharia de requisitos, 97-105
    riscos, 71-72, 130-132
    estática, 316
    sistemas, 161
    de ameaça, 136. *Veja também* risco
    usuário, 250-253
    V&V (*Verification & Validation), 348-350*
AOSD (Aspect-Oriented Software Development), 281, 509-510
    aspectos, ponto de junção e pontos de corte, 513-516
    separação de assuntos, 510-513
estímulo aperiódico, 226
APIs (*Application Programming Interfaces*) 90. *Veja também interfaces*
    reuso de aplicação de sistema, 283
apêndices, documentação de requisitos de software, 92
proteção no nível da aplicação, 481
aplicações
    arquitetura, 194-195
    sistemas de processamento de dados, 196-197
    dados de, 26
    domínios, 278
    frameworks, 282-283
    camadas, 181
    links, 408 – ligação, 270(?)
    reuso, 275
    software de, 26
    reuso de sistema, 283-289
APSE (Ada Programming Support Environment), 167
projeto de arquitetura, 51, 161-163, 215-216
    modelos de controle, 170-173
    estilos de decomposição modular, 168-170
    organização, 164-168
    arquiteturas de referência, 173-175
    na engenharia de proteção, 480-482
    seleção de sistemas, 162-164
arquiteturas, 56
    aplicações
    sistemas distribuídos, 177-179. Veja também sistemas distribuídos
    tolerantes a defeitos326-328
    camadas, 202
    MDA (*Model Driven Architecture*), 209
    sistemas de alocação de recursos, 287
erro de aritmética, 134
identificação de aspectos, 520
projeto e programação orientados a aspectos, 519-522
aspectos, 513, 514-523
    composição, 282 (aparece integração), 513, 515-516
AspectJ, 513, 521
avaliação
    sistema legados, 342
    redução de risco, 133-134
ativos
    análise, 478
    compartimentar seus, 485
    e proteção, 475-476
ATM (caixa eletrônico), 89-91
    arquitetura cliente-servidor, 180-182
    modelos de contexto, 113
    tipos de falhas, 140
    sistemas de processamento de transações, 197-202
detecção de ataque, 40, 475-476
atributos de software, 9
análise estática automatizada, 348-350
disponibilidade, 32
    sistemas críticos, 34-37
    projeto, 161-162
    métricas, 138
    ameaça a, 475
estratégias de prevenção, 72
    de riscos, 133

B

diagrama de barras, 66-69. *Veja também gerenciamento de projeto*
sistemas de processamento em lotes, 196
comportamento
    especificação formal, 152-156
    objetos de, 123-124
modelos de comportamento, 114-117
BPMN, 504
British Computer Society, 10
modelos de broadcast, 172
objetos de negócios, 184
políticas de negócios, 26
processos de negócios, 26
serviços de negócios, 497

C

modelo chamada-retorno, 170

CASE (*Computer-Aided Software Engineering*), 9
classificação, 57-59
reuso baseado em geradores, 281-282
ferramentas meta CASE, 204
processos de software, 56-59
conjuntos de ferramentas, 165
CBSE (*Component-Based Software Engineering*), 7, 43, 46-47, 291-292
composição de componentes, 299-304
modelos de, 293-298
processo, 298-299
engenharia de software, 493
controle centralizado, 170-171
classes
falhas, 140-141
hierarquias, 122
objetos, 210-212
Cleanroom
processos, 44
desenvolvimento de software, 351-353
cliente-servidor
arquiteturas, 170-183
modelo, 166
agregações, sistemas legados, 342
código de ética, 11
cores, interfaces com o usuário, 248
competência, 10
completeza, 81
verificações, 106
componentes
análise de, 46
projeto, 51
desenvolvimento para reuso, 297-298
interfaces, 294
sistemas legado, 26
métodos, 8-9
reuso, 275
teste de, 53
composição, 299-304
projetos, 520
facilidade de compreensão, 106
mau uso de computadores, 10
ciência da computação, 6
gerenciamento de sistemas de computador, 23-24
reuso de conceitos, 276
engenharia de requisitos orientada a assuntos, 517-519
informações, 516
objetos concorrentes, 212-213
confidencialidade, 10, 32
abertura, 59 (?)
ameaça a, 476
análise e resolução de conflito, 520
consistência
verificações, 106
requisitos de software, 81
construção, 55
operações de construtor, 149
contêineres, 296
modelos de contexto, 113-114
planos de contingência, 73
contratos
de manutenção, 335
desenvolvimento de software rápido, 261
modelos de controle, 170-173
sistemas de controle, 231-235
em proteção, 475-476
serviços de coordenação, 497-498
CORBA, 185-187, 295-296
projeto de sistema central, 520
manutenção corretiva, 334, 337
corrupção de programas, 39
custos, 7-8
confiança, 33
especificação formal, 146
gerenciamento de projetos, 63
reengenharia de sistema, 339, 341
COTS (comerciais de prateleira), 21
reuso do sistema da aplicação, 283-289
em projeto para proteção, 480, 486
operação Create, 150
sistemas críticos, 29-30
disponibilidade, 34-37
confiabilidade, 32-34
desenvolvimento de, 313-315
confiabilidade, 34-37
segurança, 37-38
proteção, 38-39
especificação, 129-130
tipos de, 30-32
produtos sob encomenda, 4

D

avaliação de danos, 321-324
limitação de danos, 38
aquisições de dados
projeto de software de tempo real, 236
sistemas, 235-238
banco de dados, 268
fluxo de dados
diagramas, 197
modelos, 6, 114-116
serviços de integração de dados, 174
modelos de dados, 117-120
sistemas de processamento de dados194-197
reengenharia de dados, 340
serviços de repositório de dados, 174
projeto de estrutura de dados, 51
debugging, 52, 56
decomposição, 132-133
processamento de entradas default, 319
definição de requisitos, análise e, 44
produtos a serem entregues, 65-66
entrega
desafio, 10
incremental, 47-48
recusa de serviço, 39
confiabilidade, 32-34
programação confiável, 316-321
processos de software confiáveis, 315-316
dependências, 67
configuração em tempo de implantação, 286
projetar para implantação, 485-487
projeto, 44, 50-52
de algoritmo, 51
de arquitetura. *Veja projeto de arquitetura*
de componente, 51
de estruturas de dados, 51
inspeções, 316
interfaces, 51, 240-243
mensagens, 248, 249
modelos, 218-221
orientado a objetos, 208-210
desenvolvimento rápido de software. Veja desenvolvimento rápido de software
software de tempo real. *Veja projeto de software de tempo real*
reuso, 46
reuso de software, 275-277, 281
especificação, 145
configuração em tempo de projeto, 286
detecção
defeito, 321-325
risco, 133
desenvolvimento
software Cleanroom, 351-353
componentes para reuso, 297-298
sistemas críticos, 313-315
evolucionário, 43, 45-46
integração, 46
iterativamente, 56
modelos de objetos, 120-124
processos, 292
instância de produto, 288
espiral, 49
Processo Unificado de Desenvolvimento de Software, 54
validação, 48
dicionário, 119
sistemas distribuídos
arquiteturas cliente-servidor, 179-183
arquiteturas de objetos distribuídos, 183-187
computação interorganizacional distribuída, 188-192
arquiteturas de multiprocessadores, 179
distribuição, 481
diversidade, 484
documentação. Veja também projeto de arquitetura
projeto, 220
requisitos de software, 91-93
domínios
aplicaçãoões, 278
requisitos de software, 84-85
pontos de vista, 100
arquiteturas de domínio específico, 173
alocação de memória dinâmica, 318
modelos dinâmicos, 218

E

aspecto inicial, 517
sistemas de edição, 202
recursos de grade de editor, 86
elaboração, 55
elicitação
análise de requisitos e, 50
processos de engenharia de requisitos, 98-105
processos de reparos de emergência, 338-339
propriedades emergentes de sistemas, 16-17
requisitos voláteis, 108
operação **Enter**, 150
frameworks de aplicações empresariais, 282
ambientes, 58
modelagem ERA (*Entity-Relation-Attribute*), 118
ERP (*Enterprise Resource Planning*), 5, 194, 286
tolerância a erros, 33. *Veja também sistemas críticos*
responsabilidade ética, 10-12
etnografia, 104-105, 252
avaliação, 254-255
sistemas orientados a eventos, 171
sistemas de processamento de eventos, 202-204
evolução, 6, 54, 331-332
COTS (comercial), 285
projeto, 222-223
sistemas legados, 341-345
manutenção, 334-337
processos, 337-341
dinâmica de programa, 332-334
engenharia de sistemas, 22-23
desenvolvimento evolucionário, 43, 45-46
tratamento de exceções, 319-321
exceções, 499
sistemas existentes, métodos estruturados, 124-126
desenvolvimento exploratório, 46
exposição de proteção, 475-476
extensões, 516, 519-520
requisitos externos, 83
extreme programming (XP), 263-268

F

serviço de recursos, 502
falha na proteção, 484
falhas, 29. *Veja também sistemas críticos*
classificação, 140
energia, 232
FAQ (Perguntas Mais Freqüentes), 4-10
modelo cliente-gordo, 180
defeitos. Veja também sistemas críticos
prevenção, 35
detecção e remoção, 35
software livre de defeitos, 314
tolerantes, 35, 320-328
árvores, 132, 133
estudos de viabilidade, 50
processos de engenharia de requisitos, 96-97
números em ponto flutuante, 318
métodos formais, 145, 350-353
especificação formal, 144-145
comportamento, 152-156
processo de software, 145-147
interfaces de subsistema, 15-20
formulários
desenvolvimento rápido de software, 268
especificações de linguagem estruturada, 88
frameworks
aplicações, 282-283
MVC (Model-View-Controller), 283
propriedades emergentes funcionais, 16
funcionalidade
COTS (comercial), 283
subsistemas, 19
requisitos funcionais, 18, 80-85
pipelining orientado a funções, 168-170
funções, 276

G

hierarquias de generalização, 211
relacionamentos de generalização, 121
programação baseada em geradores, 281
reuso baseado em geradores, 281-282
modelos genéricos, 173
produtos genéricos, 4
GIOP (*Generic Inter-ORB Protocol*), 187
metas, sistema, 83
recursos de grade, 86

H

hardware, 26
COTS, 22
tolerantes a defeitos, 326
confiabilidade, 16, 137-138
perigos, 131. *Veja também risco*
prevenção, 38
detecção e remoção, 38
desafio da heterogeneidade, 10
composição hierárquica, 300
hierarquias
classes, 122
generalização, 211

I

identificação
de componentes, 298
processos de evolução, 338
de objetos, 216-218
risco, 130
IDL (*Object Management Architecture*), 186
IEEE (*Institute of Electrical and Electronic Engineers*), 10
análise de impactos, 337
implementação, 44, 50-52
entrega incremental, 47-48
componentes independentes, 292
índice, 92
pontos de vista indiretos, 100
sistemas de gerenciamento de informações, 199-202
apresentação de informações, 245-249
herança, 318
modelos, 121-123
modelo de entrada-processo-saída, 196
validação de entradas, 484-485
operações de inspeção, 149
inspeções, 316
V&V (*Verification & Validation), 45-348*
bomba de insulina, 31
especificação formal, 153-156
requisitos, 88, 134
verificação de execução, 382-383
argumentos de segurança, 380-382
restrições de estado, 322
integração, 22
COTS (comercial), 283
desenvolvimento, 47
UDDI (Universal Description, Discovery and Integration), 191
integridade, ameaças à, 475
direitos sobre propriedade intelectual, 10
interação, 243-245
pontos de vista de interação, 100
envolvimento interdisciplinar, 17
interfaces, 19
entre contratados, 24-25
componentes, 293-294
projeto, 51, 240-243
gerador, 268
especificação entre objetos, 221-223
desenvolvimento rápido de software, 260
serviços, 174
especificação, 90-91, 147-152
prototipação baseada na Internet, 254
interoperabilidade, 285
computação interorganizacional distribuída, 188-192
interrupções, 318
modelos orientados a interrupções, 172
entrevista, 101-102
risco intolerável, 131
alarme de intrusos, 232
IOR (*Interoperable Object Reference), 186*
interação
desenvolvimento, 5, 56
RUP (Rational Unified Process), 54
processos de software, 47-49

J

Java
applets, 182
processo de BuildingMonitor, 234
tratamento de exceções, 319-320
tolerantes a defeitos, 320-325
informações protegidas, 317
versões de tempo real, 227
modelo JINI, 190

pontos de junção, 282, 513-516
modelo, 513, 515

L

projeto de interface de serviço, 499
linguagens
especificação formal, 147
IDL (*Object Management Architecture*), 186
Java. *Veja* Java
OCL (*Object Constrain Laguage*), 302
design, 279-281
sistemas de processamento, 204-205
SQL (*Structured Query Language*), 182
UML (*Unified Modeling Language*), 8, 102
programação visual, 253-254
WSDL (*Web Services Description Language), 191*
reuso em larga escala, 162
modelos em camadas, 167-168
sistemas legados, 27
camadas
aplicações, 180
arquitetura, 201
operação Leave, 150
sistemas legados, 25-26
evolução, 341-344
serviços de, 501-502
leis de Lehman, 332-334
LIBSYS, 81, 83
sistemas de contabilidade, 86
arquitetura, 200
modelos de dados, 120
requisitos de domínio, 84-85
interfaces, 245
modelagem de comportamento de objetos, 123-124
requisitos, 100
reuso de software, 281
XP (extreme programming), 263
ciclos de vida, 44
avaliação de riscos de ciclo de vida, 477, 479-480
tempo de vida, software, 278
redundância de estrutura de dados ligados, 324-325
links, 268
serviço de login, 502
operação Lookup, 150

M

facilidade de manutenção, 33
projeto, 162
manutenção, 44. *Veja também resolução de problemas*
evolutiva, 334-337
desenvolvimento rápido de software, 260
serviço de, 502
gerenciamento
processos, 230-231
projetos. Veja gerenciamento de projetos
desenvolvimento rápido de software, 260
requisitos, 55, 316
gerenciamento de requisitos, 107-110
segurança, 134-136
teste, 316
modelo gerenciador, 170
requisitos obrigatórios de software, 87
MDA (Model Driven Architecture), 209
Mensagens
projeto, 248-249
serviços, 174
ferramentas meta CASE, 204
métodos, 8-9
ágeis. *Veja* métodos ágeis
componentes, 9
V&V (*Verification & Validation), 350-353*
métricas
avaliação de interfaces, 254-255
confiabilidade, 138-139
middleware, 185, 292
frameworks de integração, 282
marcos, 65
estratégias de minimização, 72
código móvel, 182
modelos
projeto de arquitetura, 164
CBSE (engenharia de software baseada em componentes), 293-298
verificação, 316
projeto, 218-221
objetos distribuídos, 184
MDA (Model Driven Architecture), 209
processos, 6
processos de software, 43-47
métodos estruturados, 51
modelos de sistema. *Veja modelos de sistema*
arquitetura de sistemas, 20
modelos de uso, projeto orientado a objetos, 215
modificação de requisitos , 46
estilos de decomposição modular, 168-169
modularização, 340
monitoração, 63
projeto de software de tempo real, 231-235
risco, 73-74
operação Move150, 152
MTTF (tempo médio para falha), 138
MTTR (tempo médio para reparo), 138
herança múltipla, 123
arquiteturas de multiprocessadores, 179
MVC (Model-View-Controller), 246, 283

N

projeto de nomes, 520
namespaces, 495
linguagem natural, 87
navegação de interfaces, 243-245
redes
arquiteturas cliente-servidor, 179-183
protocolos, 187
neutralização, 40
aquisição de dados de fluxo de nêutrons, 236
propriedades emergentes não funcionais, 16
requisitos de software não funcionais, 80-85, 139-141
programação não preemptiva, 231
programação em N versões, 326

O

definição de objetivos, 48
modelos de objeto, 120-124
decomposição orientada a objetos, 168-169
projeto orientado a objetos, 208-210
classes, 210-213
processo de, 213-223
objetos
agregação, 123
modelagem de comportamento, 123
concorrente, 212-213
de objetos distribuídos, 183-187
identificação, 216-218
reuso, 275
especificação, 147
padrão Observer, 280
OCL (*Object Constrain Laguage*), 302
OMG (*Object Management Group*), 185
OO (orientados a objetos), 8
sistemas operacionais, 229-231
operação, 44-45
tipo abstrato de dados, 149-150
incompatibilidade, 301
inconclusão, 301
estado, 117
confiabilidade, 16, 137-138
processos operacionais, 24-25
ORBs (*Object Request Brokers*), 172, 186
requisitos organizacionais, 83
organização de sistema, 164-168
organizações, 23-24
OWL-S, 501

P

programação em pares, 267-268
paralelismo, 318
incompatibilidade de parâmetros, 301
agrupar, 19
padrões
projeto, 279-281
Observer, 280
gerenciamento de pessoal, 23-24
manutenção evolutiva, 334
desempenho, 16
projeto, 161
estímulos periódicos, 226
planejamento, 48
modelos algorítmicos de custos, 419-422
gerenciamento de projeto, 63-65

planejamento de engenharia de requisitos, 108-110
risco, 71-73
teste, 316
V&V (*Verification & Validation), 344*
proteção no nível de plataforma, 481
plataforma, reuso de software, 278
POFOD (probabilidade de falha sob demanda), 138
ponto de corte, 513, 514-515
ponteiros, 318
políticas, negócio, 26
falha de energia, 232
sistemas p2p (ponto a ponto), 188-189
precisão, 32
previsão, manutenção, 336-337
programação preemptiva, 231
avaliação de riscos preliminar, 477-479
detecção preventiva de defeitos, 322
software crítico primário de segurança, 37
interfaces de procedimentos, 90
serviços de processos, 497
processos, 6
negócio, 26
CBSE (Component-Based Software Engineering), 298-299
Cleanroom, 44
debbuging, 52, 56
software confiável, 315-316
desenvolvimento, 292
reparos de emergenciais, 339
de evolução, 337-341
especificação formal, 145-147
gerenciamento, 230-231
modelos, 6-7
projeto orientado a objetos, 213-222
operação, 24
organizacionais, 24-25
paradigmas, 43
planejamento, 63
desenvolvimento de protótipo, 272
Rational Unified Process, 43
engenharia de requisitos, 95-97. *Veja também* processo de engenharia de requisitos
RUP (*Rational Unified Process*), 97
software. *Veja* processos de software
Processo Unificado de Desenvolvimento de Software, 54
projeto de interface com o usuário, 249-250
XP (*Extreme Programming),* 263-268
aquisição
desenvolvimento, 24
sistemas, 284
custos de desenvolvimento de produto, 7
produção, 5
linhas de produto, 283, 285-289
requisitos de produtos, 83
responsabilidade profissional, 10-12
inspeção de programa, 316, 346-348
aprimoramento da estrutura do programa, 340
gerenciamento de projetos, 61-62
atividades, 62-63
planejamento, 63-65
cronogramas, 65-69
propriedades, 18
propostas
obtenção de requisitos, 99-104
elaboração, 62
informações protegidas, 317
proteção, 481
prototipação
de interface, 253-254
desenvolvimento rápido de software, 270-273
requisitos, 106
de sistemas, 250

Q

consultas, 182
questionários, 254

R

RAD (*Rapid Application Development Tecnique*), 268-270
desenvolvimento rápido de software, 260-262
métodos ágeis,262-263
prototipação, 270-273
RAD (*Rapid Application Development Tecnique*), 268-270
XP (*Extreme Programming*), 263-268
verificações de realismo, 106
projeto de software de tempo real, 225-227
sistemas de controle, 231-235
aquisições de dados, 235-237
monitoração, 231-235
RTOS (*Real Time Operating System),* 229-231
projeto de sistema, 227-229
reconhecimento em capacidade de sobrevivência, 488
proteção no nível de registro, 481
recuperação
blocos, 327
projetar para, 485
de defeitos, 325
em capacidade de sobrevivência, 487
recursão, 318
redundância, 325
em engenharia de proteção, 484
TMR (Triple-Modular Redundancy), 326
reengenharia, 339-341
arquiteturas de referência, 173-175
confiabilidade, 32
sistemas críticos, 34-36, 138-141
métricas, 138-139
remoção de riscos, 133
facilidade de reparos, 33
geradores de relatórios, 268
modelo de repositório, 165-166
representações, 112
requisitos. *Veja também requisitos de software*
análise, 44
gerenciamento de mudança, 110
obtenção, 99-105
elicitação e análise, 50
gerenciar, 56, 316
modificação, 46
proteção, 137
especificação, 50
validação, 50
processos de engenharia de requisitos, 79, 95-96
análises, 97-105
elicitação, 97-105
estudos de viabilidade, 97
gerenciamento, 107-110
validação, 105-106
resistência em capacidade de sobrevivência, 488
sistemas de alocação de recursos, 201, 287
sistemas de gerenciamento de recursos, 199-202
detecção retroativa de defeitos, 322
reuso
desenvolvimento de componentes para, 297-298
projeto, 46
larga escala, 162
serviços como componentes, 493-496
software. *Veja* reuso de software
programação visual, 269
engenharia reversa, 340
revisões de requisitos, 106
risco
análise, 71-72
avaliação, 48, 133-134, 477-478
classificação, 130-132
especificação de sistema crítico, 130-134
identificação, 70
gerenciamento de, 476-480
monitoração, 73
planejamento, 72-73
gerenciamento de projeto, 69-73
reengenharia de sistemas, 339
redução, 133-134
ROCOF (taxa de ocorrência de falhas), 138
modelos de papel/ação, 7
RTOS (*Real Time Operating System*), 229-231
esquema RUN, 154
RUP (*Rational Unified Process*), 43, 54-56
estudos de viabilidade, 97

S

programação segura, 318-319
segurança, 32
sistemas críticos, 37-38

especificação de sistema crítico, 134-136
projeto, 162
espalhamento, 511-512
cenários, engenharia de requisitos, 102
programação
não preemptiva/preemptiva, 231
desenvolvimento de projetos65-69
reuso de software, 277-278
esquemas
RUN, 154
Z, 152
escopo, 17
abordagem dirigida a scripts, 253
software crítico secundário de segurança, 37
proteção, 32
sistemas críticos, 38-39
especificação de sistema crítico, 136
projeto, 162
política, 483
requisitos, 137
facilidade de uso, 484
engenharia de proteção, 473-474
projeto de arquitetura, 480-482
conceitos, 475-476
gerenciamento de riscos, 476-477
capacidade de sobrevivência de sistemas, 486-489
modelos semânticos de dados, 118
separação de assuntos, 281, 510-513
modelos de seqüência, 219
composição seqüencial, 300
servidores
arquiteturas cliente-servidor, 179-183
objetos concorrentes, 212
identificação de serviço candidato, 497-498
engenharia de serviços, 496-502
identificação de candidato, 497-498
implementação e implantação, 501
projeto de interface, 499-500
serviços de sistemas legados, 501-502
engenharia de software orientada a serviços, 491-493
desenvolvimento de, 502-507
componentes reusáveis, 493-496
engenharia de serviços, 496-502
teste, 506-507
projeto e implementação de workflow, 504-506
arquitetura de sistema orientada a serviços, 187-191, 491
serviços
modelos de componentes, 297
de interfaces, 175
modelos de, 190
ponto único de falha, 483
SOAP (*Simple Object Access Protocol*), 191, 492
engenharia social, 484
sistemas sociotécnicos, 15
engenharia de software, 4
desafios, 10
custos, 7-8
FAQs, 4-10
métodos, 8-9
orientado a serviços, 493-494
processos de software, 42-43
atividades, 49-54
CASE (*Computer-Aided Software Engineering*), 56-59
iteração, 47-49
modelos, 43-47
RUP (*Rational Unified Process), 54-56*
confiabilidade de software, 17, 137-141
requisitos de software, 80
documento de, 91-93
de domínio, 84-85
funcional e não funcional, 80-85
especificação de interfaces, 91
SRS (Software Requirements Specification), 91
requisitos de sistema, 87-89
de usuários, 85-87
reuso de software, 275-277
frameworks de aplicações, 282-283
reuso de sistema de aplicação, 283-289
reuso baseado em geradores, 281-282
técnicas, 277-279
conversão de código-fonte, 340
especialização, reuso de software, 286-289
especificação, 6
abstrata, 51
sistemas críticos, 129
formal. *Veja* especificação formal
interface, 89-90, 221-223
linguagem natural, 87
objetos, 233
de requisitos, 50
de proteção, 136-137
de software, 49-50
SRS (*Software Requirements Specification*), 91
linguagem estruturada, 88-89
desenvolvimento em espiral, 48-49
SQL (*Structured Query Language*), 182
banco de dados, 268
SRS (*Software Requirements Specification*)), 91
stakeholders, 98
comunicação, 161
padrões, componentes, 292
statecharts, 116
modelos de máquina de estados, 116-117, 219
análise estática, 316
V&V (*Verification & Validation*), 348-351
modelos estáticos, 218
Análise Estruturada, 8
especificações de linguagem estruturada, 88-91
métodos estruturados, 8-9. 51
modelos de sistema, 124-126
subsistemas, 15
modelos de broadcast, 172
desenvolvimento, 21-22
especificação formal, 147-152
identificar, 19
modelos, 219
apoio
COTS (comercial de prateleira), 285
Software de, 26
pesquisas, 254
capacidade de sobrevivência, 33
análise de sobrevivência de sistemas, 486, 487
análise de sistema, 161
contexto de sistema, projeto orientado a objetos, 215-216
falhas de sistema, 30. *Veja também sistemas críticos*
frameworks de infra-estruturas de sistemas, 282
modelos de sistema, 112-113
modelos de comportamento, 114-117
modelos de contexto, 113-114
modelos de dados, 117-120
modelos objetos, 120-124
métodos estruturados, 124-126
reengenharia de sistema, 339-341
requisitos de sistemas, 87-89
engenharia de sistemas, 17-18
contratados, 24
em comparação a engenharia de software, 5-6
desativação, 23
projeto, 19-20
evolução, 22-23
modelagem, 20-21
definição de requisitos, 18-19
desenvolvimento de subsistema, 21-22

T

especificação tabular de cálculo, 90
embaralhamento, 512, 514
análise de tarefas, 252
tarefas
duração, 67
serviços de gerenciamento, 174
sistemas técnicos baseados em computadores, 15
teste, 44, 53
planejamento, 316
XP (extreme programming), 265-267
geração de casos de teste, 106
modelo cliente-magro, 180
análise de ameaças, 136, 475-476
cliente-servidores de três camadas, 182
prototipação throwaway, 45
tempo certo, 32
TMR (*Triple-Modular Redundancy*), 326
ferramentas
meta CASE, 204
apoio de método, 125

processos de software, 57
rastreabilidade, 107-108
sistemas de processamento de transações, 197-202
transição, 55
resolução de problemas, 44. *Veja também* sistemas críticos (não aparece)
CBSE (Component-Based Software Engineering), 291-293
debugging, 52, 56
classificação de falhas, 140
tolerantes a defeitos, 320-325
interfaces, 242-249
interoperabilidade, 285
reuso de software, 276
arquitetura cliente-servidor de duas camadas, 181

U

UD (*Universal Design*), 243
UDDI (*Universal Description, Discovery, and Integration*), 191, 492, 493, 501, 504
UML (*Unified Modeling Language*), 8, 103
vetores não limitados, 318
Processo Unificado de Desenvolvimento de Software , 54
casos-de-uso, engenharia de requisitos, 102-104
interfaces com o usuário. *Veja também interfaces*
projeto, 240-243
serviços, 174
usuários
interação do, 243-245
requisitos de software, 85-87
serviços de utilidades, 497

V

validação, 6, 52-53
em sistemas orientados a aspectos, 521-523
desenvolvimento, 48
desenvolvimento rápido de software, 260
requisitos, 50, 105-106
verificações de validade, 106
facilidade de verificação, 106
verificação
em sistemas orientados a aspectos, 521-523
requisitos de software não funcionais, 83
gerenciamento de requisitos, 107
abordagens orientadas a pontos de vista para engenharia de requisitos, 100-101, 517
programação visual
linguagens, 253
reuso, 269
requisitos voláteis, 107-108
método de engenharia de requisitos VOLERE, 87
vulnerabilidade
prevenção, 39
e proteção, 1475, 480
V&V (*Verification & Validation*) 341-344
análise estática automatizada, 348-350
métodos formais, 350-353
inspeções, 345-348
planejamento, 344-345

W

modelo em cascata, 7, 43-45
WSDL (*Web Services Description Language*), 191, 493, 494-495, 500
interfaces baseadas na Web, 244. *Veja também* interfaces
workbenches, 57
workflows
modelos, 6
RUP (Rational Unified Process), 56

X

XP (Extreme Programming), 263-267

Y

YAWL, 504

ÍNDICE ONOMÁSTICO

A
Abbott, 217
Abowd, 256
Abts, 285
Ackroyd, 24
Adams, 36
Addy, 289
Alexander, 279
Ambler, 262
Anderson, 474
Andrews, 506
Appelrath, 194
Appleton, 280
Arango, 107
Arnold, 27
Arthur, 338
Avizienis, 327

B
Baker, 283
Balk, 283
Baniassad, 520
Banker, 337
Barker, 119
Barnard, 347
Bass, 161, 163, 164, 175
Beale, 256
Beck, 48, 59, 93, 217, 262, 263, 273
Beedle, 262
Belady, 332
Bell, 131
Bentley, 98, 252
Berczuk, 280
Berghel, 39, 484
Bernstein, 179, 192, 206
Bieman, 269, 272
Biggerstaff, 281
Birrer, 515
Bisbal, 27
Bishop, 474
Blevins, 295
Boehm, 48, 49, 72, 263, 273, 285, 297, 337, 341
Booch, 8, 51, 121, 124, 209, 223
Borchers, 280
Bosch, 162, 166, 175, 206
Bott, 12
Bourne, 269
Bracket, 197
Braun, 289
Brazendale, 131
Brerton, 192
Brilliant, 327
Brinch-Hansen, 228
Brook, 27
Brooks, 13, 74
Brown, 174
Budgen, 192
Burns, 231, 238
Butler, 281
Buxton, 165, 167

C
Cabrera, 506
Chapman, 30, 129, 145, 156
Checkland, 16, 24
Chen, 119
Chess, 480
Chikofsky, 339
Clarke, 520
Clement, 510
Clements, 164, 170, 175
Coad, 121, 209, 217
Cobb, 352, 353
Cockburn, 262
Codd, 119
Cohen, 148
Coleman, 13
Colyer510, 521
Constantine, 51
Constantinos, 523
Cooling, 228, 238
Coulouris, 177, 192
Councill, 293, 304
Crabtree, 252
Crnkovic, 304
Cross, 339
Cumings, 40
Cunningham, 217
Cusamano, 277
Czarnecki, 281

D
Davis, 79, 92
Dehbonei, 145
DeMarco, 8, 74, 115, 263
Diaper, 251
Dijkstra, 228, 318, 522
Diller, 152
Dix, 241, 256
Dollimore, 192
Dorfman, 94
Douglass, 228, 238
Draper, 253
Dunn, 40
Dyer, 36

E
Easterbrook, 107, 145, 517
Eaton, 13
Eisenecher, 281
Elliott, 240
Ellison, 33, 40, 487
Elrad, 281
Erl, 492, 493, 497, 501
Erlikh, 331
Evans, 350, 480
Ewald, 295

F
Fagan, 345, 346
Fayad, 282
Felsing, 262
Finkelstein, 100, 517
Finlay, 256
Firesmith, 137, 480
Foster, 190
Fowler, 103
Frewin, 344
Fromme, 172
Fuggetta, 57, 59
Fujiwara, 324

G
Galin, 354
Gamma, 279, 283, 289
Gane, 51
Garlan, 164, 172, 175, 205, 285
Ghezi, 12
Gilb, 345, 346, 347
Goguen, 152
Goldberg, 217, 245
Gomma, 228
Gordon, 269, 272
Gotterbarn, 10, 13
Gradecki, 521
Grady, 346
Graham, 209, 345, 346, 347
Griss, 277
Gudgin, 191
Guttag, 148

H
Hailpern, 354
Hall, 30, 129, 145, 146, 156
Halstead, 337
Hammer, 119
Hardin, 227
Harel, 116, 220, 228
Harker, 108
Harkey, 179
Harold, 195
Harvey, 132
Hatton, 344
Hayes, 152
Heath, 105
Heineman, 293, 304
Helm, 289
Heninger, 88, 92
Highsmith, 262
Higuera-Toledano, 227
Hnich, 304
Hoare, 228
Hofmeister, 162
Horswill, 199
Huang, 325
Huff, 12, 56
Hughes, 105, 252
Hull, 119
Humphrey, 344, 347

I
Issarny, 227

J
Jackson, 27, 87, 124
Jacky, 145, 152
Jacobsen, 102, 121, 124, 209, 223, 298, 511, 516, 519, 520
Jahanian, 132
Jeffries, 262
Johnson, 282, 289
Jones, 152, 352
Jonsson, 304

K
Kafura, 337
Katz, 522
Kavantzas, 506
Kazman, 170, 175
Kedia, 283
Kennedy, 240
Kiczales, 282, 510
Kifer, 206
Kindberg, 192
King, 119
Kintala, 325
Kit, 344
Kiziltan, 304
Kleppe, 209, 302
Knight, 325
Knuth, 279
Kotonya, 81, 91, 94, 10, 111, 517
Kreger, 190
Krutchen, 54, 59, 124
Kumaran, 190
Kume, 37

L
Laddad, 510, 521
Lam, 480
Lamping, 247
Laprie, 30, 327
Larman, 97, 223, 262
Larochelle, 350, 480
Larus, 480
Laudon, 12
Lehman, 332, 333
Leveson, 37, 130, 132, 327
Lewis, 206, 345
Lezeiki, 521
Lientz, 334
Linger, 33, 44, 352, 353, 354
Liskov, 148
Lister, 74
Littlewood, 36
Livshits, 480
Lomow, 493
Lovelock, 189
Luff, 105
Luqi, 156
Lutz, 37, 269, 351
Lyu, 328

M
Mandrioli, 12
Martin, 12, 105, 280
Matsumoto, 277
McCabe, 337
McConnell, 74
McDougall, 188
McGraw, 480, 483
McGuffin, 165
MCIlroy, 275
McLeod, 119
Means, 195
Mejia, 145
Mellor, 116, 217
Meyer, 303
Mili, 289, 297
Miller, 10, 12, 199, 241
Mills, 36, 44, 345, 352, 353
Mitschele-Thiel, 352
Mok, 132
Mullery, 100
Mumford, 24
Musciano, 240

N
Nakajo, 37
Neil, 145
Neilsen, 254
Neumann, 40
Newcomer, 493
Ng, 511, 516, 519, 520
Nii, 166
Nilsen, 227
Norman, 253
Nosek, 334
Nuseibeh, 517

O
O'Connor, 281
O'Leary, 286
Oram, 188, 192
Orcero, 350
Orfali, 179
Ostoff, 243
Ould, 59, 74
Ousterhout, 269

P
Palmer, 262
Palvia, 334
Parnas, 314
Pedrycz, 12
Perrow, 38
Peters, 12
Peterson, 132
Pfaff, 244
Pfarr, 283
Pfleeger, 39, 474
Plakosh, 345
Poore, 354
Pope, 292
Potter, 152
Powell, 354
Pradhan, 324
Preece, 256
Preiser, 243
Price, 347
Prieto-Díaz, 107
Pritchard, 185
Prowell, 44, 145, 352
Pullum, 326, 328

R
Randell, 327
Reddy, 337
Redmill, 135
Reiss, 172, 283
Rittel, 19
Ritter, 194
Robertson, 87, 94, 111
Robinson, 51, 124, 209, 217
Robson, 245
Rogers, 256
Rogerson, 10, 12
Rowland, 12
Royce, 44
Rubin, 217
Rumbaugh, 8, 51, 54, 121, 124, 209, 223

S
Sametinger, 295, 296
Santhanam, 354
Saron, 124
Sarson, 51
Sawyer, 517
Schmidt, 282
Schneier, 40, 483
Scholes, 24
Schwaber, 262
Scott, 103, 287
Seacord, 345
Selby, 44, 345, 353
Sharp, 256
Shaw, 164, 175, 205
Shlaer, 217
Shneiderman, 241, 243, 247
Silberschatz, 228
Skonnard, 191
Snyder, 253
Sommerville, 81, 91, 94, 100, 105, 111, 517
Spafford, 39, 484
Spivey, 145, 152, 345
Stal, 190
Stapleton, 262, 273
Stevens, 27
Stolzy, 132
Storey, 132, 314
Suchman, 105, 252
Swanson, 334
Swartz, 16
Szyperski, 185, 293, 298, 304

T
Tanenbaum, 228
ten Hagen, 244
ter Hofstede, 504
Thayer, 15,17, 27, 94
Thompson, 480
Tracz, 283
Trammell, 354
Turner, 190, 192

V
Van Slack, 346
ven der Aalst, 504
Viega, 480, 483
Viller, 105
Vlissides, 289

W
Walker, 172
Wall, 269
Wang, 295
Ward, 116
Warner, 302
Warren, 342, 345
Webber, 19
Weigers, 94
Weinberg, 267
Weinreich, 295, 296
Weiss, 241
Wellings, 231, 238
Westmark, 486
Wheeler, 483
White, 15, 504
Whittaker, 480
Williams, 268
Wirfs-Brock, 217, 282
Wordsworth, 44, 129, 145, 152, 351
Wosser, 277

X
Xu, 327

Y
Yacoub, 289
Yourdon, 51, 121, 209, 217

Z
Zimmermann, 167